Theoretical

In this section you will complete some exercises related to discussing symmetry groups of molecules. Consider a function $\phi : G \to H$, where G and H are groups, and suppose that ϕ is a homomorphism, i.e. $\phi(g_1 g_2) = \phi(g_1)\phi(g_2)$.

1. Show that $\phi(1_G) = 1_H$.

The kernel of ϕ, denoted $\ker(\phi)$, is the collection of elements of G that ϕ maps to 1_H, i.e.,

$$\ker(\phi) = \{g \in G \mid \phi(g) = 1_H\}.$$

2. Show that $\ker(\phi)$ is a subgroup of G.
3. Show that ϕ is 1-1 if and only if $\ker(\phi) = \{1_G\}$, i.e., the only element in the kernel of ϕ is the identity element of G.

Suppose X is a G-set, and that Y is a G-invariant subset of X, which means that $g(Y) = Y$ for every $g \in G$. For a given $g \in G$, consider the map

$$r[g](y) = g \cdot y.$$

4. For a fixed g, show that $r[g]$ is invertible, and write a formula for its inverse $r[g]^{-1}$.

Recall from HW1 that a permutation of Y is a 1-1 and onto function from Y to itself, and that S_Y is the group of all permutations of Y.

5. Show that the map $r : G \to S_Y$, given by $g \mapsto r[g]$, is a homomorphism of groups.

Numerics

For this assignment, please hand in (for example) a Python program, e.g., a `.py` file, that completes the following tasks. Use Python comments to answer questions that aren't explicit computations. Make sure your program runs!

n-grams

An n-gram is a sequence of n consecutive words in a sequence of text. In this problem we will process and count n-grams from the novel Moby Dick by Herman Melville. The purpose of this exercise is to familiarize you with loading and parsing text, working with arrays and loops, and analyzing program efficiency. You can download Moby Dick at `http://www.math.wisc.edu/~dynerman/moby_dick.txt`.

1. Load Moby Dick into Python. See the Python commands `open` and `readlines`.
2. Split the text into a list of words. See the Python string method `split`.
3. Write a function `ngram( n, k, text )` that returns the k most frequent n-grams in the list of words `text`.
4. Print the 10 most frequent 3, 4 and 5-grams in Moby Dick.

Algorithm Runtime

Estimating the runtime of algorithms is useful when working with large datasets. An algorithm that runs reasonably on a small dataset might stall on a large one. In other words, the algorithm might scale poorly.

We can estimate the runtime of an algorithm by counting the number of arithmetic operations an algorithm performs. For example, consider the linear search algorithm for finding the position of an element x in a list.

```
def linear_search( list , x ):
    for i in range(0,len(list)):
        if list[i] == x:
            return i

    return -1    # Return a position of -1 if x is not in list
```

Analyzing this algorithm, we see that, in the worst case, the for loop will traverse the entire list, performing one operation (the `==` comparison) on each element. We say that `linear_search` has linear run-time: if we call this function on a list of n elements, then the function `linear_search` will perform n operations.

5. Estimate the run-time of your `ngram( n, k, text )` in terms of the length of `text`. **Hint:** Examine your function and see how many times each block of code processes `text`. Is a block of code linear? Quadratic? Exponential? Estimate the overall run-time by summing the run-time of each block. When estimating run-time, we only report the highest order (slowest) part of this sum.

Remark: Generally, algorithms that have polynomial run-time in the length of the input are considered efficient. Algorithms with exponential (or worse) run-time are considered inefficient.

Numerical integration and vectorized operations

Many common operations in scientific computing involve performing the same operation on many pieces of data. The efficiency of such algorithms can often be improved if they are implemented using so-called vectorized operations.

A quadrature rule is a numerical approximation of a definite integral. We write

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i). \tag{1}$$

Where w_i are certain weights and $x_i \in [-1, 1]$. The n-point Gaussian Quadrature rule is a collection of w_i and x_i that make the above numerical approximation exact when f is a polynomial of degree $\leq 2n - 1$.

Consider the integral

$$\int_{-1}^{1} e^{x^2} \cos(x)\,dx. \tag{2}$$

6. Generate the 50-point Gaussian Quadrature weights w_i and points x_i. See `polynomial.legendre.leggauss` in `numpy`.

7. With these quadrature weights and points, use a for loop to compute the sum in [Equation 1] for the integral [Equation 2].

8. Time how long your computation takes. See the Python function `timeit`.

In the next few exercises we will vectorize the above for loop.

9. Populate two vectors `w` and `x` with the quadrature weights and points you generated above. These should be `(50,1)` vectors.

10. Populate a `(50,1)` vector with $f(x_i)$. See `vectorize` in `numpy`.

11. Compute the approximation [Equation 1] using these vectors. Hint: You can multiply vectors element-wise in one operation. Then, see `sum` in `numpy`.

12. Time how long your vectorized computation takes. Compare with your previous implementation.