# Homework 1

April 14, 2020

## 1 Homework 1 - More fun with interference

### 1.1 Preamble

The purpose of this homework is to further familiarize you with interference and how we can use it to solve certain problems efficiently.

Each task has a certain number of points, for a total of **135pt**. You only need **100pt** to get full marks on this assignment.

For each task you have to write some **Python code** but also a **brief explanation** of why your solution works. You will be graded on both (for each task, 70% of the points will be for the code and 30% for the written explanation). The written explanation can be either in the form of comments in your code or as a separate description in a text file (or both).

To turn in your homework, submit a zip file with your solutions on Moodle.

The deadline is: **April 20, 23:59 PST**.

#### 1.1.1 Frequently Asked Questions

**Q.** If my solution produces the correct output on all the tests do I get full marks?
**A.** Not necessarily. The tests are there to help you check your implementation and you're free to add more if you wish or ignore them altogether. Your solution will be graded based on whether the implementation you wrote is correct and solves the problem (this is why it's useful to add the explanation of how your solution works).

**Q.** If I get over 100pt are the bonus points added to the next homework assignment?
**A.** No. Anything above 100 is truncated at 100.

**Q.** Do I get partial credit for an incomplete solution?
**A.** Yes.

**Q.** What if I don't write the code but just a written description of the solution?
**A.** You will get partial credit that's at most 30% of the total number of points for that task, since that's what's allotted for written explanations.

**Q.** Can we collaborate?
**A.** Yes, you are encouraged to discuss the homework with the other students, however your solution should be your own.

**Q.** Do we need to use typing? I.e. do we need to specify the types for every function we write?
**A.** No. The typing is just indicative to help you, you're free to ignore it.

[1]:
```
from interf import *
```

## 1.2 (25pt) Task 1: interf1 from interf2

In the lectures we've seen two types of functions that can perform interference: `interf1` and `interf2`. With `interf1` the output is just a boolean value depending on whether the interference was strongly constructive or strongly destructive. More precisely, for some function $f : D \to \{-1, +1\}$ on some domain, $D$, we were computing

$$S = \frac{1}{N}\left|\sum_{x \in D} f(x)\right|$$

with $N$ being a suitably chosen normalization constant, and returning `True` if $S \geq 2/3$ and `False` if $S \leq 1/3$.

With `interf2` the output is a string that's sampled according to a probability distribution obtained through interference. There we had as parameters not just $f$ but also a transformation $T(x, y)$ and we were computing

$$Pr(y) = \left|\sum_{x \in D} f(x)T(x, y)\right|$$

suitably normalized. The output is $y$, sampled from $D$ according to this probability distribution.

We can see that `interf2` is a more general form of interference and the purpose of this task is to show that fact.

Your task is to implement `interf1` using `interf2`. Your implementation should have a number of calls to `interf2` that is at most some polynomial in $log(|D|)$ and should behave like `interf1` with, say, 99% probability. In other words, your implementation should produce the same results as `interf1` 99% of the time. Allowed operations include:

1. Computing the size of $D$. So things like `len(domain)` are ok (in practice this operation takes time linear in $|D|$, but if you imagine that $D$ is presented as an efficient function that you can query for elements in $D$, then you should know the range that you can query).
2. Querying $D$. So things like `domain[i]` are ok (of course the number of such queries must be at most $poly(log(|D|))$).
3. Defining a new domain as a function of $D$ (e.g. `newDomain = list(zip(domain, domain))` or `newDomain = list(map(myFunction, domain))`, as long as `myFunction` is an efficient function).
4. When defining the $T$ used by `interf2`, you are allowed to call $f$ inside $T$ (as long as the number of calls to $f$ that $T$ makes is at most $poly(log(|D|))$).

What's not allowed is computing things like $\sum_{x \in D} f(x)$ and generally aggregating results about more than $poly(log(|D|))$ elements in $D$. You need to make use of `interf2` to do any computationally expensive operation.

We'll consider two cases.

#### 1.2.1 (20pt) Binary strings case

To make things easier (and similar to the lectures), for this case you can assume the domain, $D$, is the set of all $n$-bit (or `numBits`-bit) binary strings, $\{0, 1\}^n$.

#### 1.2.2 (5pt) Generic case

For this case you should assume that `domain` is just some generic set of elements. Of course, a solution to this case subsumes the previous case.

Complete the code block below (and specify if your solution is generic or works only for the binary strings case).

```
[3]: def interf1from2(fun: Callable[[Any], int], domain: List[Any], norm: int) ->␣
     ↪bool:
         # your code here
         return False
```

Make sure to also explain why your implementation works. You can test your function with the examples from the Deutsch-Jozsa problem that we saw in Lecture 1. These have been added in the file *dj.py*.

### 1.3 (20pt) Task 2: k-query interference (interf2k)

With both `interf1` and `interf2` we are essentially performing a single call to the black-box oracle. We can, of course, call these functions multiple times, which entails performing multiple queries (like we did when solving Simon's problem). However, doing this means that we are not relating the different queries when performing interference. If we want our model to be as general as possible, we should allow for multi-query interference as well. In other words, we should be performing interference on the results of multiple queries simultaneously (we refer to these as *coherent queries* in the quantum setting).

More precisely, for a function $f : D \to \{-1, +1\}$, we would like to sample a string $y \in D$, according to the following probability distribution

$$Pr(y) = \left| \sum_{x_1, x_2, \ldots x_k \in D} f(x_1)T_1(x_1, x_2)f(x_2)T_2(x_2, x_3)\ldots T_{k-1}(x_{k-1}, x_k)f(x_k)T_k(x_k, y) \right|$$

suitably normalized. Here, $x_i \in \{0, 1\}^n$ and $k$ denotes the number of coherent queries being performed. Also, the notation

$$\sum_{x_1,x_2,...x_k \in D} \equiv \sum_{x_1 \in D} \sum_{x_2 \in D} \cdots \sum_{x_k \in D}$$

Is just a short-hand for repeated summation. Each $x_i$ is ranging over the entire set $D$ (independently of the other $x_i$'s). Note how the queries are combined together using the transformations $T_i$. We can now see that that `interf2` is the $k = 1$ query version of this.

To be as general as possible, we can also make each query be performed to a potentially different oracle function. In other words, for oracle functions $f_1, f_2, ...f_k : D \to \{-1, +1\}$, we would like to sample a string $y \in D$, according to the suitably normalized distribution

$$Pr(y) = \left| \sum_{x_1,x_2,...x_k \in D} f_1(x_1)T_1(x_1,x_2)f_2(x_2)T_2(x_2,x_3)...T_{k-1}(x_{k-1},x_k)f_k(x_k)T_k(x_k,y) \right|$$

Note that if we were to take $f_i = f$, we recover the version from above.

Write a function, denoted `interf2k`, that achieves this task. Do this by completing the following code block:

```
[6]: def interf2k(funList: List[Callable[[Any], int]], transfList:␣
     ↪List[Callable[[Any, Any], float]], domain: List[Any]) -> Any:
         # your code here
         return []
```

Here `funList` is the list of $k$ functions $f_i$, $i \leq k$ for which we have black-box query access and `transfList` is the list of transformations, $T_i$.

### 1.4 (20pt) Task 3: Forrelation

Another interesting problem that we can solve efficiently using interference, but for which classical algorithms require exponentially-many queries to solve is known as *Forrelation* (you can read more about it here). In this problem, you're given oracle access to two functions

$$f : \{0,1\}^n \to \{-1,+1\}$$

$$g : \{0,1\}^n \to \{-1,+1\}$$

The problem is to determine whether $f$ is correlated with the *Fourier transform* of $g$ or not (hence the name). Since the functions are essentially Boolean (just outputting -1 and +1 instead of 0 and 1), the Fourier transform is taken over $\mathbb{Z}_2$, i.e.

$$F(x,y) = \frac{1}{2^{n/2}}(-1)^{x \cdot y}$$

Thus, forrelation asks whether the following quantity

$$\Phi(f,g) = \frac{1}{2^{3n/2}} \left| \sum_{x,y\in\{0,1\}^n} f(x)(-1)^{x\cdot y} g(y) \right|$$

is greater than 2/3 or less than 1/3, promised that one of these is the case.

Write an efficient algorithm to determine this, in the interference model. In other words, you should implement an algorithm that runs in polynomial time and makes polynomially-many queries in $n$ to $f$ and $g$, which makes use of the `interf` functions (you can use any combination of `interf1`, `interf2` and `interf2k`). Do this by completing the code block:

```
[7]: def forrelation(f: Callable[[BinStr], int], g: Callable[[BinStr], int]) -> bool:
         # your code here
         return False
```

This function takes $f$ and $g$ as arguments. The number of bits, $n$, on which each function acts, is a global variable, as in previous implementations. The function has to output `True` if $\Phi(f,g) \geq 2/3$ and `False` if $\Phi(f,g) \leq 1/3$. In between these cases, the output should be random.

You can test your implementation with the functions provided in *forrelation.py*.

## 1.5 (30pt) Task 4: Recursive Fourier Sampling

In the lectures, we used `interf2` to solve the Bernstein-Vazirani problem. You're now going to use that approach to solve a more general problem known as *Recursive Fourier Sampling* (RFS). RFS essentially consists of recursive embeddings of the Bernstein-Vazirani (BV) problem. To make things simpler, we'll first consider a 2-level recursion of the problem so that we understand what "recursive embeddings of BV" means. Then we'll tackle the more general problem. Before that, let's recall the BV problem and consider its decision version (see these notes for a good description of RFS and the decision version of Bernstein-Vazirani).

### 1.5.1 Decision-version of Bernstein-Vazirani

Recall that you were given oracle access to $f : \{0,1\}^n \to \{0,1\}$ defined as $f(x) = s \cdot x$, for some $s \in \{0,1\}^n$. The task was to find $s$ with a small number of queries to $f$. Classically, we require $n$ queries, whereas in the interference model we saw that the problem can be solved using a single query. What would a decision version of the problem look like? Suppose that instead of having to output $s$, we need to output some function $g(s)$, where $g : \{0,1\}^n \to \{0,1\}$. This $g$ is a *hard-core bit* (or hard-core predicate) of $s$. This just means that determining $g(s)$ should be as hard as determining $s$ itself. While we're not going to prove this, we're going to take $g$ to be the following function

$$g(s) = \begin{cases} 0, \text{ if } |s| \ mod \ 3 = 0, \\ 1, \text{otherwise} \end{cases}$$

Here, $|s|$ denotes the Hamming weight of $s$ (the number of 1's in the binary representation of $s$). The decision version of BV is now: given oracle access to $f(x) = s \cdot x$, output $g(s)$.

### 1.5.2 (15pt) 4A: Two-level recursion

We'll now consider a "recursed" version of BV. Suppose that we're given access to $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, such that

$$f(x, y) = s_x \cdot y$$

where the strings $s_x \in \{0,1\}^n$ are chosen so that $g(s_x) = x \cdot s$, for some $s \in \{0,1\}^n$. The task is to find $s$, or alternatively $g(s)$.

Why is this a two-level version of BV? If we fix $x$ and consider $f_x(y) = f(x, y)$, solving the BV problem for $f_x$ yields $s_x$. Knowing $x$ and $s_x$, we can then compute $g(s_x)$, which we know is $x \cdot s$. This is exactly like one query to the original $f$ in the regular BV problem. We had to solve an instance of BV to get one query to another instance of BV.

Your task is to solve this two-level BV problem in the interference model (you can use whichever `interf` function you like, but you might want to consider using `interf2k` since that will make things simpler). Complete the code-block below with your implementation:

[5]:
```python
def bv2(f: Callable[[BinStr, BinStr], bool]) -> int:
    # your code here
    return 0
```

Test your implementation with the functions in *bv2.py*.

### 1.5.3 (15pt) 4B: k-level recursion

We now take the idea from 3A and apply it recusively $k$ times, for some $k > 0$. Suppose you're given oracle access to $f : \{0,1\}^{n\times k} \to \{0,1\}$ such that

$$f(x_1, x_2, ...x_k) = s_{x_1,x_2,...x_{k-1}} \cdot x_k$$

where $s_{x_1,x_2,...x_{k-1}} \in \{0,1\}^n$ is such that

$$g(s_{x_1,x_2,...x_{k-1}}) = s_{x_1,x_2,...x_{k-2}} \cdot x_{k-1}.$$

Similarly, each $s_{x_1,x_2,...x_{k-2}} \in \{0,1\}^n$ is such that

$$g(s_{x_1,x_2,...x_{k-2}}) = s_{x_1,x_2,...x_{k-3}} \cdot x_{k-2}$$

and so on. At the end of this recursion we have that $g(s_{x_1}) = s \cdot x_1$. Find $g(s)$.

Once again, you need to solve this in the interference model. Complete the code-block below with your implementation:

[8]:
```
def rfs(f: Callable[[List[BinStr]], bool], k: int) -> int:
    # your code here
    return 0
```

Note that **rfs** takes two arguments: the function $f$ for which we're given black-box access, as well as the number $k$, that specifies how many $n$-bit strings $f$ will take as an argument. Test your implementation with the functions in *rfs.py*.

### 1.6 (30pt) Task 5: Period-finding

In the lectures we saw how we can solve Simon's problem using interference. There, we were given access to a function $f : \{0,1\}^n \to \{0,1\}^n$ that was periodic over $n$-bit strings, with period $s \in \{0,1\}^n$, $s \neq 0^n$. In other words, we had $f(x) = f(x \oplus s)$. The goal was to find $s$ using as few queries to $f$ as possible. We solved the problem in the interference model using a linear (in $n$) number of queries to $f$.

We're now going to consider a generalized version of this problem. You are now given access to a function $f : \mathbb{Z}_d \to \mathbb{Z}_d$, where $\mathbb{Z}_d$ is the field of integers modulo $d$ and you are promised that $f(x) = f(x + s)$, where $s \in \mathbb{Z}_d$, $s \neq 0$ and the addition is performed modulo $d$. In other words, the function is periodic modulo $d$. Your goal is to find $s$ using a number of queries to $f$ that scales like $O(poly(\log(d)))$.

Complete the function below that finds the period of $f$ with high probability and test your implementation with the functions in *periodfinding.py*.

[17]:
```
def periodFinder(f: Callable[[int], int], d: int) -> int:
    # your code here
    return 0
```

The main way to approach this problem is to try to generalize the algorithm we have for Simon's problem to integers mod $d$ rather than binary strings. We considereded functions whose output is $+1$ or $-1$ when using `interf1` and `interf2`. Of course, we don't need to limit ourselves to these values, since it's possible to perform interference with other values as well. The most natural extension to $d$ values that add up to 0 is to consider the $d$ roots of unity:

$$R_d = \{e^{2\pi i k/d} \mid 0 \leq k \leq d-1\}.$$

This also gives us a nice extension of the Fourier transform to the field of integers modulo $d$, which is just:

$$F_d(x,y) = e^{\frac{2\pi i}{d} xy}$$

up to normalization, and where $x, y \in \mathbb{Z}_d$ and the multiplication $xy$ is performed modulo $d$. As you might expect, this should substitute the mod 2 Fourier transform that we've used up to this point $(F_2(x,y) = (-1)^{x \cdot y})$.

With these notions, we can now pass functions to `interf1`, `interf2` and `interf2k` that output complex numbers, rather than integers (we can change the type signatures of those functions to complex numbers).

See slides 45 to 50 from here for a brief explanation of how to generalize Simon's algorithm to period-finding.

## 1.7 (10pt) Task 6: Create your own problem

By this point you're hopefully convinced of the usefulness of interference in solving certain problems efficiently. Specifically, two things should be clear:

1. In this oracle setting where we're trying to minimize the total number of calls to a specific function, there are problems that can be solved with interference using exponentially fewer calls than with any deterministic or probabilistic algorithm.
2. Interference is not a magic bullet that can be used to solve any problem more efficiently than with deterministic or probabilistic algorithms. Instead, these problems have some special structure that makes them ammenable to an interference-based solution.

Now it's your turn to be creative. Come up with your own problem that admits an efficient interference-based solution but that seems intractable without interference. Specifically, you need to do the following:

1. Define a problem in the query model. In other words, specify a function $f$ (or multiple functions), for which you are given black box access and define a problem relative to this function. This problem should not be one that has been already covered in class or one from the existing literature. You can, however, take an existing problem and modify it to come up with a different one.
2. Give an argument for why any deterministic or probabilistic algorithm cannot solve the problem in polynomial time. You don't have to provide a formal proof, a qualitative argument is sufficient.
3. Write down your algorithm for the problem in the interference model. You are recommended to use the `interf` functions that we defined (including `interf2k`), however you can also define other interference functions if you think they are useful (though you should explain why these new functions are still performing interference).