CHICAGO JOURNAL OF THEORETICAL COMPUTER SCIENCE 2012, Article 06, pages 1–10
*http://cjtcs.cs.uchicago.edu/*

# Interactive proofs with efficient quantum prover for recursive Fourier sampling

Matthew McKague[*]

*Received: August 24, 2011; revised: September 18, 2012; published: September 21, 2012.*

**Abstract:** We consider the recursive Fourier sampling problem (RFS), and show that there exists an interactive proof for RFS with an efficient classical verifier and efficient quantum prover.

**Key words and phrases:** interactive proofs, quantum computing, recursive Fourier sampling

## 1 Introduction

The Recursive Fourier Sampling (RFS) problem is an oracle based decision problem that was proposed by Bernstein and Vazirani in [3]. Historically, RFS was the first problem that showed a relativized separation between BQP and P.

Along with their definition of RFS, Bernstein and Vazirani also proved a lower bound for the classical query complexity and an upper bound for the quantum query complexity, establishing the P versus relativized BQP separation. Later Aaronson [1] proved matching lower bounds for the quantum query complexity of RFS. Johnson [7] extended and improved these results, including bounds on the polynomial degree of RFS. Hallgren and Harrow [6] subsequently generalized the recursive structure of the problem to allow a super-polynomial speedup, relative to an oracle, based on randomly generated quantum circuits.

Bernstein and Vazirani also showed that RFS is not in NP. Aaronson [1] claims that the proof can be extended to show that RFS is not in MA, and it has been conjectured by Vazirani and others that it is not even in PH. This gives an oracle relative to which BQP is not in MA, and hope that a relativized separation may be found between BQP and PH.

[*]Thanks to Scott Aaronson for his comments and suggestion for VERIFIER2. This work is funded by the Centre for Quantum Technologies, which is funded by the Singapore Ministry of Education and the Singapore National Research Foundation.

DOI: 10.4086/cjtcs.0006

In the remainder of this section we recall the definition of the recursive Fourier sampling problem and the optimal classical and quantum algorithms. Our main contribution is in section 2, where we give an algorithm for an interactive proof for RFS and show how the standard quantum algorithm for RFS can be adapted to be used as an efficient prover for the interactive proof.

## 1.1 The recursive Fourier sampling problem

We begin by defining a type of tree. Let $n, l$ be a positive integers and consider a symmetric tree where each node, except the leaves, has $2^n$ children, and the depth is $l$. Let the root be labeled by $(\emptyset)$. The root's children are labelled $(x_1)$ with $x_1 \in \{0,1\}^n$. Each child of $(x_1)$ is, in turn, labelled $(x_1, x_2)$ with $x_2 \in \{0,1\}^n$. We continue until we have reached the leaves, which are labelled by $(x_1, \ldots, x_l)$. Thus each node's label can be thought of as a path describing how to find the node from the root.

Now we add the Fourier component to the tree. We begin by fixing an efficiently computable function $g : \{0,1\}^n \to \{0,1\}$.[1] With each node of the tree $(x_1, \ldots, x_k)$ we associate a "secret" string $s_{(x_1,\ldots,x_k)} \in \{0,1\}^n$. These secrets are promised to obey

$$g\left(s_{(x_1,\ldots,x_k)}\right) = s_{(x_1,\ldots,x_{k-1})} \cdot x_k \tag{1.1}$$

for $k \geq 1$, and the inner product taken modulo 2. (Here we take $s_{(x_1,\ldots,x_{k-1})}$ to mean $s_{(\emptyset)}$ if $k = 1$.) In this way, each node's secret encodes one bit of information about its parent's secret.

Now suppose that we are given an oracle $A : (\{0,1\}^n)^l \to \{0,1\}$ which behaves as[2]

$$A(x_1, \ldots, x_l) = g\left(s_{(x_1,\ldots,x_l)}\right). \tag{1.2}$$

Note that $A$ works for the leaves of the tree *only*. Our goal is to find $g\left(s_{(\emptyset)}\right)$. This is the recursive Fourier sampling problem (RFS).

At this point we wish to discuss the recursive nature of the RFS problem. First, note that the subtree rooted at any node obeys the same promises as the whole tree. Thus each subtree defines an RFS problem. (The subtree rooted at a leaf is a trivial problem, where the oracle just returns the solution.) Thus we have a methodology for solving RFS problems: solve subtrees in order to determine information about the root's secret, then calculate the secret. Solving subtrees uses the same method, except when we reach a leaf, where the oracle is used instead. This is a type of top down recursive structure, where the tree is built from a number of subtrees with the same structure.

Another way of viewing the tree is from the bottom up. Note that if we remove all the leaf nodes, truncating the tree, then the remaining tree still obeys all the required promises. The oracle no longer returns relevant information, however. This problem can be solved by building a new oracle using the old one. The new oracle, given $(x_1, \ldots, x_{l-1})$, calculates the secret $s_{(x_1,\ldots,x_{l-1})}$ by accessing the old oracle. The new oracle then returns $g(s_{(x_1,\ldots,x_{l-1})})$. An oracle constructed in this way behaves exactly as the old oracle does, but for the truncated tree. The tree can be truncated again to depth $l-2$ and another new oracle is built on the previous one. The result is a recursive algorithm that eventually solves the RFS problem.

[1]For concreteness $g(s)$ may be taken to be 0 if $h(s) \cong 0 \mod 3$ and 1 otherwise, where $h(x)$ is the Hamming weight of $s$ (number of 1s in $s$).

[2]In fact the values $s_{(x_1,\ldots,x_l)}$ are not necessary, since we can only ever learn $g(s_{(x_1,\ldots,x_l)})$. However, including them in the definition eliminates special cases at the level $l$.

With a little thought, it is easy to see that both pictures of the recursive structure of the RFS problem are essentially equivalent. Indeed, solving a subtree rooted at $(x_1,\ldots,x_k)$ just means returning $g(s_{(x_1,\ldots,x_k)})$, which is what the oracle for the tree truncated at level $k$ does. However, each picture can be useful in understanding the structure of algorithms.

## 1.2 Classical solution

Now let us consider the following solution to the recursive Fourier sampling problem in the classical setting. To calculate $g\left(s_{(\emptyset)}\right)$ we must first find $s_{(\emptyset)}$. To do so, let us define $1_j$ to be the $n$-bit string with a 1 in the $j$th position and 0 elsewhere. Define $(s_{(\emptyset)})_j$ to be the $j$th bit of $s_{(\emptyset)}$, which is given by $s_{(\emptyset)} \cdot 1_j$. These values are given by the solution to the RFS problem defined on the subtree rooted at $(x_1)$ with $x_1 = 1_j$. After determining these values for $j = 1 \ldots n$ we have determined $s_{(\emptyset)}$. Hence we obtain the following algorithm:

**Algorithm 1** (RFS)**.** Input: oracle $A$, subroutine $g$, $l,k$, $(x_1,\ldots,x_k)$

```
1. If $k = l$ then return $A(x_1,\ldots,x_l)$
2. For $j = 1 \ldots n$
    (a) Set $(s)_j = \mathrm{RFS}(A,g,l,k+1,(x_1,\ldots,x_k,1_j))$
3. Return $g(s)$
```

**Lemma 1.1.** *$RFS(A,g,l,0,())$ returns $g(s_{(\emptyset)})$.*

*Proof.* We proceed by induction. We claim that for $0 \le k \le l$

$$\mathrm{RFS}(A,g,l,k,(x_1,\ldots,x_k)) = g(s_{(x_1,\ldots,x_k)}). \tag{1.3}$$

For $k = l$ this is true by the definition of the oracle.

Now suppose that $0 \le k < l$. By induction, step (a) sets $(s)_j = g\left(s_{(x_1,\ldots,x_k,1_j)}\right)$ which is promised to equal $s_{(x_1,\ldots,x_k)} \cdot 1_j$. Thus $s = s_{(x_1,\ldots,x_k)}$ and the function returns $g(s_{(x_1,\ldots,x_k)})$, as claimed. Calling $RFS(A,g,l,0,())$ then returns $g\left(s_{(\emptyset)}\right)$.

□

The query complexity of this algorithm is $n^l$, since each call to $RFS$ calls itself $n$ times, and the depth of recursion is $l$.

This solution is due to Bernstein and Vazirani [3], who also gave a matching lower bound on the number of queries, so the query complexity is $\Theta(n^l)$.

## 1.3 Quantum solution

We now consider the analogous quantum problem, where the oracle $A$ allows quantum access according to

$$A\,|x_1\rangle \ldots |x_l\rangle\,|y\rangle = |x_1\rangle \ldots |x_l\rangle\,\left|y \oplus g(s_{(x_1,\ldots,x_l)})\right\rangle. \tag{1.4}$$

In addition, we have an efficient quantum algorithm which calculates $g$ as

$$G|s\rangle|y\rangle = |s\rangle|y \oplus g(s)\rangle. \tag{1.5}$$

The main idea behind the algorithm is to use the fact that $H^{\otimes n}$ transforms $|y\rangle$ into $\sum_x (-1)^{x\cdot y}|x\rangle$ and vice versa. We use phase feedback and a call to $A$ to create the state $\sum_{x_l}(-1)^{s_{(x_1,\dots,x_{l-1})}\cdot x_l}|x\rangle$, then apply $H^{\otimes n}$ to obtain $\left|s_{(x_1,\dots,x_{l-1})}\right\rangle$. After calculating $g(s_{(x_1,\dots,x_{l-1})})$, we uncompute $\left|s_{(x_1,\dots,x_{l-1})}\right\rangle$ to disentangle this register from other registers.

**Algorithm 2** (QRFS)**.** Input: oracle $A$, subroutine $G_j$, $l$, $k$, quantum registers $\mathcal{X}_1,\dots,\mathcal{X}_k$, $\mathcal{Y}$.

```
1. If $k = l$ then apply $A$ to $(\mathcal{X}_1 \dots \mathcal{X}_l)$ and $\mathcal{Y}$, then return.
2. Introduce ancilla $\mathcal{X}_{k+1}$ in the state $\frac{1}{\sqrt{2^n}}\sum_{x_{k+1}\in\{0,1\}^n}|x_{k+1}\rangle_{\mathcal{X}_{k+1}}$.
3. Introduce ancilla $\mathcal{Y}'$ in state $\frac{1}{\sqrt{2}}(|0\rangle_{\mathcal{Y}'} - |1\rangle_{\mathcal{Y}'})$.
4. Call $QRFS(A,G,l,k+1,(\mathcal{X}_1\dots\mathcal{X}_{k+1}),\mathcal{Y}')$
5. Apply $H^{\otimes n}$ on register $\mathcal{X}_{k+1}$.
6. Apply $G$ to $\mathcal{X}_{k+1}$ and $\mathcal{Y}$.
7. Apply $H^{\otimes n}$ on register $\mathcal{X}_{k+1}$.
8. Call $QRFS(A,G,l,k+1,(\mathcal{X}_1\dots\mathcal{X}_{k+1}),\mathcal{Y}')$
9. Discard $\mathcal{X}_{k+1}$ and $\mathcal{Y}'$.
10. Return
```

**Lemma 1.2.** *$QRFS(A,G,l,0,(),|0\rangle)$ returns $\left|g(s_{(\emptyset)})\right\rangle$.*

*Proof.* To analyze the correctness of the algorithm we proceed by induction with the hypothesis that for each $0 \le k \le l$, $QRFS(A,G,l,k,(\mathcal{X}_1\dots\mathcal{X}_k),\mathcal{Y})$ applied to $|x_1\rangle\dots|x_k\rangle|y\rangle$ gives

$$|x_1\rangle\dots|x_k\rangle\left|y\oplus g\left(s_{(x_1,\dots,x_k)}\right)\right\rangle. \tag{1.6}$$

(The output for general input states is determined by linearity.) This is true for $k = l$ by definition of the oracle.

Now let $0 \le k < l$. We introduce the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ in step 3 in order to use phase kickback. By hypothesis, then, step 4 introduces phase of $(-1)^{g(s_{(x_1,\dots,x_{k+1})})}$, which is the same as $(-1)^{s_{(x_1,\dots,x_k)}\cdot x_{k+1}}$. The state after step 4 is thus

$$\frac{1}{\sqrt{2^n}}\sum_{x_{k+1}}(-1)^{s_{(x_1,\dots,x_k)}\cdot x_{k+1}}|x_1\rangle\dots|x_{k+1}\rangle|y\rangle|-\rangle. \tag{1.7}$$

After step 5 the state then becomes

$$|x_1\rangle\dots|x_k\rangle\left|s_{(x_1,\dots,x_k)}\right\rangle|y\rangle|-\rangle. \tag{1.8}$$

Step 6 changes $|y\rangle$ to $\left|y \oplus g\left(s_{(x_1,\ldots,x_k)}\right)\right\rangle$. Next, steps 7 and 8 uncompute the $\mathcal{X}_{k+1}$ register, so it and the $\mathcal{Y}'$ register are returned to their original state when they are discarded in step 9. Thus we obtain the required output state.

From the inductive hypothesis, we see that $QRFS(A,G,l,0,(),\mathcal{Y})$ applied to $|0\rangle$ gives $\left|g\left(s_{(\emptyset)}\right)\right\rangle$. $\square$

Note that if we apply $QRFS(A,G,l,k,(\mathcal{X}_1 \ldots \mathcal{X}_k),\mathcal{Y})$ to $|x_1\rangle \ldots |x_k\rangle |y\rangle$, and stop after step 5, we obtain $\left|s_{(x_1,\ldots,x_k)}\right\rangle$ in the $\mathcal{X}_{k+1}$ register. This is equivalent to solving the RFS problem defined by the subtree rooted at $(x_1,\ldots,x_k)$. Thus we can also efficiently calculate any $s_{(x_1,\ldots,x_k)}$.

This algorithm is due to Bernstein and Vazirani [3]. The quantum query complexity of this algorithm is $O(2^l)$, since two recursive calls are made, and the depth is $l$. Aaronson [1] gave a matching lower bound.

### 1.4 Complexity implications

In the previous sections we have kept both $n$, the size of the bit strings, and $l$, the depth of the tree. Typically, RFS is considered with $l = \log_2 n$. In this case we obtain query complexities of $\Theta(n^{\log_2 n})$ classically, and $O(n)$ quantumly. Hence we obtain the relativized separation of $\mathsf{BQP} \not\subseteq \mathsf{P}$. It is also for this value of $l$ that RFS $\notin$ $\mathsf{MA}$.

## 2 Interactive proof

Suppose now that we have, in addition to the oracle $A$ and classical computing resources, access to a prover $P$ who has more powerful computing resources. We have seen that for the choice $l = \log_2 n$ there is no way of efficiently computing the solution to the recursive Fourier sampling problem directly. In fact, since RFS is not in $\mathsf{NP}$, we cannot even efficiently verify a solution if it is given (along with a short proof.) We will show, however, that by *interacting* with $P$ we can efficiently compute the solution, or detect if $P$ is giving false information. That is, RFS is in $\mathsf{IP}$.

It is not surprising that RFS is in $\mathsf{IP}$, indeed $\mathsf{BQP} \subseteq \mathsf{IP}$ in the unrelativized world. However, $\mathsf{IP}$ is defined with an computationally unbounded prover, and here we will see that for RFS it suffices to have an efficient quantum prover.

### 2.1 Classical verifier 1

Looking back at the original classical solution to RFS in Algorithm 1 we see that there are two steps: find $s_{(x_1,\ldots,x_k)}$ and calculate $g(s_{(x_1,\ldots,x_k)})$. The difficult part is finding $s_{(x_1,\ldots,x_k)}$, so we can instead ask $P$ to do this for us. Since we do not trust $P$, we should perform some type of test to see whether $P$ has really given us the correct value of $s_{(x_1,\ldots,x_k)}$.

Suppose that $P$ gives us a string $s'_{(x_1,\ldots,x_k)}$. In principle we can detect the case $s'_{(x_1,\ldots,x_k)} \neq s_{(x_1,\ldots,x_k)}$ by instead looking at whether $s'_{(x_1,\ldots,x_k)} \cdot x_{k+1} = s_{(x_1,\ldots,x_k)} \cdot x_{k+1}$ for a randomly chosen $x_{k+1} \in \{0,1\}^n$. If $s'_{(x_1,\ldots,x_k)} \neq s_{(x_1,\ldots,x_k)}$ then $s'_{(x_1,\ldots,x_k)} \cdot x_{k+1} \neq s_{(x_1,\ldots,x_k)} \cdot x_{k+1}$ for half of all strings $x_{k+1} \in \{0,1\}^n$. If we check for $c$ independently chosen strings $x_{k+1}$ then we will fail to detect $P$'s deception with probability $2^{-c}$.

Since we have $s'_{(x_1,\ldots,x_k)}$, we can calculate $s'_{(x_1,\ldots,x_k)} \cdot x_{k+1}$ readily, but how do we find $s_{(x_1,\ldots,x_k)} \cdot x_{k+1}$? We use recursion: we ask $P$ for $s_{(x_1,\ldots,x_{k+1})}$ and again perform the test. After the recursion is deep enough, we can query the oracle directly to find $s_{(x_1,\ldots,x_{l-1})} \cdot x_l = A(x_1,\ldots,x_l)$.

**Algorithm 3** (VERIFIER)**.** Input: Oracle $A$, Prover $P$, subroutine $g$, total number of levels $l$, current level $k$, queries $(x_1,\ldots x_k)$.

```
1. If $k = l$ then return $A(x_1,\ldots,x_l)$
2. Query $P$ for $s'_{(x_1,\ldots,x_k)}$
3. Repeat 3 times:
    (a) Randomly choose $x_{k+1} \in \{0,1\}^n$
    (b) Set $a = \text{VERIFIER}(A,P,g,l,k+1,(x_1,\ldots,x_{k+1}))$
    (c) If $a \neq s'_{(x_1,\ldots,x_k)} \cdot x_{k+1}$ then abort
4. Return $g(s'_{x_1,\ldots,x_k})$
```

We must also specify what the behaviour of $P$ should be. When $P$ receives a query $(x_1,\ldots,x_k)$, it should return a string $s'_{(x_1,\ldots,x_k)} \in \{0,1\}^n$. For an honest $P$ this should be equal to $s_{(x_1,\ldots,x_k)}$.

Clearly there exists a prover $P$ which always returns $s_{(x_1,\ldots,x_k)}$ when queried: $P$ can solve a subtree of the full RFS problem, rooted at $(x_1,\ldots,x_k)$ to find $s_{(x_1,\ldots,x_k)}$. For such a prover, we may perform VERIFIER$(A,P,g,l,0,())$ to obtain $g\left(s_{(\emptyset)}\right)$, which is the answer to the RFS problem.

**Lemma 2.1.** *Let $P$ be a prover that always returns $s'_{(x_1,\ldots,x_k)} = s_{(x_1,\ldots,x_k)}$ when queried. Then*

$$\mathit{VERIFIER}(A,P,g,l,0,()) = g(s_{(\emptyset)}). \tag{2.1}$$

*Proof.* We proceed via induction, as in previous proofs. We claim that for $0 \le k \le l$

$$\text{VERIFIER}(A,P,g,l,k,(x_1,\ldots,x_k)) = g(s_{(x_1,\ldots,x_k)}). \tag{2.2}$$

This is true for $k = l$ from the definition of the problem and line 1. For $k < l$ we see from line 4 that the claim holds for this choice of $P$ so long as the algorithm does not abort in line (c).

Now suppose that $0 \le k < l$. By induction, step (b) sets $a = g(s_{(x_1,\ldots,x_{k+1})}) = s_{(x_1,\ldots,x_{k-1})} \cdot x_k$ by the definition of the problem. Again, by the choice of $P$ this is always equal to $s'_{(x_1,\ldots,x_{k-1})} \cdot x_k$ and the algorithm never aborts.

□

Our next concern is what the algorithm does when interacting with a prover $P$ that does not return correct strings. The next lemma says that the algorithm returns an incorrect result with low probability; the rest of the time the algorithm aborts. Hence we are protected from a malicious $P$.

**Lemma 2.2.** *For any $P$, the probability that VERIFIER$(A,P,g,l,0,())$ does not abort and returns a value not equal to $g(s_{(\emptyset)})$ is at most $\frac{1}{4}$.*

*Proof.* Let $p_k$, $0 \leq k \leq l$ be the probability that VERIFIER does not abort and returns a value that is *not* equal to $g(s_{(x_1,\ldots,k_k)})$. To be precise, we should specify what $P$ does, since $p_k$ can depend on $P$'s behaviour. Let us then take the maximal $p_k$ over all choices of $P$, which will give us an upper bound for any particular $P$.

We proceed by induction with the hypothesis that $p_k \leq 1/4$. For $k = l$ this true since $A(x_1,\ldots x_l) = g(s_{(x_1,\ldots,x_l)})$ so $p_l = 0$.

Now suppose that $0 \leq k < l$. There are two cases. First, $P$ returns $s'_{(x_1,\ldots,x_k)}$ such that $g(s'_{(x_1,\ldots,k_k)}) = g(s_{(x_1,\ldots,k_k)})$. In this case VERIFIER always returns the correct value, so $p_k = 0$. Note that VERIFIER may still abort, since $s'_{(x_1,\ldots,x_k)}$ may not equal $s_{(x_1,\ldots,x_k)}$, or the recursion may return an incorrect result.

Now consider the case where $P$ returns some $s'_{(x_1,\ldots x_k)}$ such that $g(s'_{(x_1,\ldots x_k)}) \neq g(s_{(x_1,\ldots x_k)})$. If the algorithm does not abort in line (c) then one of two things happened: either $s'_{(x_1,\ldots x_k)} \cdot x_{k+1} = s_{(x_1,\ldots x_k)} \cdot x_{k+1}$ for this choice of $x_{k+1}$, which happens with probability $\frac{1}{2}$, or $s'_{(x_1,\ldots x_k)} \cdot x_{k+1} \neq s_{(x_1,\ldots x_k)} \cdot x_{k+1}$ and at the same time the recursion in (b) returns an incorrect value. The latter two events occur together with probability at most $\frac{p_{k+1}}{2}$. Thus, for each of the three repetitions, the probability of not aborting is at most $\frac{1}{2}(1+p_{k+1})$ and the overall chance of not aborting is

$$p_k \leq \frac{1}{2^3}\left(1+p_{k+1}\right)^3. \tag{2.3}$$

By the induction hypothesis, $p_{k+1} \leq \frac{1}{4}$. Then

$$p_k \leq \frac{1}{8}\left(1+\frac{1}{4}\right)^3 = \frac{125}{512} \leq \frac{1}{4}. \tag{2.4}$$

Using induction, we obtain the result for $0 \leq k \leq l$. In particular, $p_0 \leq 1/4$, which is the desired result. □

**Corollary 2.3.** *VERIFIER*$(A,P,g,l,0,())$ *solves the recursive Fourier sampling problem with completeness 1 and soundness* $1/4$.

The algorithm uses $3^l$ queries to the oracle and fewer than $3^l$ to the prover. With the choice $l = \log_2 n$ this is polynomial in $n$.

## 2.2 Classical Verifier 2

While the previous verifier can be seen as a type of parallel repetition, we can also perform serial repetition, gaining some advantage in running time. This solution is due to by Scott Aaronson (personal communication).

**Algorithm 4** (VERIFIER2)**.** Input: Oracle $A$, Prover $P$, subroutine $g$, total number of levels $l$, current level $k$, queries $(x_1,\ldots x_k)$.

1. If $k = l$ then return $A(x_1,\ldots,x_l)$
2. Query $P$ for $s'_{(x_1,\ldots,x_k)}$

3. Randomly choose $x_{k+1} \in \{0,1\}^n$

4. Set $a = \text{VERIFIER2}(A, P, g, l, k+1, (x_1, \ldots, x_{k+1}))$

5. If $a \neq s'_{(x_1,\ldots,x_k)} \cdot x_{k+1}$ then abort

6. Return $g(s'_{x_1,\ldots,x_k})$

**Lemma 2.4.** *VERIFIER2$(A, P, g, l, 0, ())$ solves the recursive Fourier sampling problem with completeness 1 and soundness* $1 - 2^{-l}$.

*Proof.* Clearly an honest prover can always provide $s = s'$ and the test in step 5 always passes, so the verifier accepts with probability 1.

Let $p_k$, $0 \leq k \leq l$ be the minumum probability that at level $k$ VERIFIER2 either aborts or returns $g(s_{x_1,\ldots,x_k})$. Thus $p_l = 1$ by definition of $A$.

Now suppose that for some $k$, $p_{k+1} \geq 2^{-k}$. If prover provides $s'_{x_1,\ldots,x_k} \neq s_{x_1,\ldots,x_k}$ then with probability $1/2$, $x_{k+1}$ is chosen in step 3 such that $x_{k+1} \cdot s'_{x_1,\ldots,x_k} \neq x_{k+1} \cdot s_{x_1,\ldots,x_k}$. In this case, in step 4, with probability $p_{k+1}$, we either abort or are given $g(s_{x_1,\cdots,x_k})$ in which case we detect the cheating prover and abort in step 5. Thus if $g(s'_{x_1,\ldots,x_k}) \neq g(s_{x_1,\ldots,x_k})$ the protocol will abort with probability at least $p_k \geq 2^{-k-1}$.

We find, then, that $p_0 \geq 2^{-l}$. This indicates that a cheating prover causes the verifier to accept with probability no greater than $1 - 2^{-l}$. □

Using this protocol, together with serial repetition, we may achieve constant soundness through $O(2^l)$ repetitions of the protocol. For $l = \log n$ this again gives a polynomial complexity.

### 2.3 Quantum prover

Although the class IP is usually defined for a computationally unbounded prover, it happens that for RFS and the verifier presented above, the prover can be an efficient quantum prover. That is, if the prover is quantum then it need only make a polynomial number of queries to the oracle.

As mentioned in section 1.3, algorithm 2 makes at most $2^l$ queries to find $s_{(\emptyset)}$ and can be adapted to find $s_{(x_1,\ldots,x_k)}$ using at most this number of queries. We can thus readily adapt algorithm 2 to create a prover for the above classical verifier. The total number queries that the prover will make is less than $3^l 2^l \leq n^{2.58}$. For VERIFIER2 the number of queries is less than $2^{2l} = n^2$.

## 3 Discussion

Recently there has been interest in interactive proofs in a quantum context and, most relevant here, interaction between quantum provers and classical verifiers. Work in this direction began with Mayers and Yao [10], and Magniez et al. [9] who showed how to classically verify (with certain assumptions) the operation of quantum apparatus, including entire circuits. Recently, Aharonov et al. [2] and Broadbent et al. [4] (with later improvements by Fitzsimons and Kashefi [5]) considered verifiers with limited quantum capabilities. As well, Broadbent et al. suggest that their result can be modified for classical verifiers by introducing an additional entangled prover. Ideally we would like to show that every problem in BQP has an interactive proof with a single efficient quantum prover and a classical verifier (let us call the class of such problems $\mathsf{IP}_{\mathsf{BQP}}$.) Unfortunately, self-testing and the Broadbent et al. protocol both rely fundamentally upon non-local properties of quantum theory, making the techniques unsuitable to the single-prover scenario.

In this context, the current work is interesting because it shows that there is an oracle relative to which $\mathsf{IP}_{\mathsf{BQP}} \not\subseteq \mathsf{MA}$[3]. Since $\mathsf{BQP} \subseteq \mathsf{PSPACE} = \mathsf{IP}$ an interactive proof exists for every problem in BQP, but the only known construction, due to Shamir [11] and Lund et al. [8], uses a PSPACE prover. The interest for the current result, then, is the fact that the interactive proof has an efficient quantum prover.

Another interesting aspect of this work is the fact that the structure of the interactive proof is not informed by the structure of the prover but arises naturally from the structure of the problem. This is in contrast to results in [9] and [4], which essentially analyze the quantum provers in action to verify correct operation. This may indicate that a different methodology from these partial results is necessary, or at least useful, in order to reduce the number of provers down to one.

**Acknowledgements** Thanks to Scott Aaronson for his comments and suggestion for VERIFIER2. This work is funded by the Centre for Quantum Technologies, which is funded by the Singapore Ministry of Education and the Singapore National Research Foundation.

## References

[1] Scott Aaronson. Quantum lower bound for recursive Fourier sampling. *Quantum Information and Computation*, 3:165, 2003. 1, 5

[2] Dorit Aharonov, Michael Ben-Or, and Elad Eban. Interactive proofs for quantum computations, October 2008. 8

[3] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. 1, 3, 5

[4] Anne Broadbent, Joseph Fitzsimons, and Elham Kashefi. Universal blind quantum computation. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 517–526, Los Alamitos, CA, USA, 2009. IEEE Computer Society. 8, 9

[5] Joseph F. Fitzsimons and Elham Kashefi. Unconditionally verifiable blind computation, March 2012. 8

[6] Sean Hallgren and Aram Harrow. Superpolynomial speedups based on almost any quantum circuit. In Luca Aceto, Ivan Damgård, Leslie Goldberg, Magnús Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, volume 5125 of *Lecture Notes in Computer Science*, pages 782–795. Springer Berlin / Heidelberg, 2008. 1, 9

[3]Note that the more general result of Hallgren and Harrow [6] does not have this property, since the oracle in their construction can be used to verify a certificate.

[7] Benjamin Johnson. *Upper and Lower Bounds for Recursive Fourier Sampling*. PhD thesis, University of California Berkeley, 2008. 1

[8] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 2 –10 vol.1, oct 1990. 9

[9] Frédéric Magniez, Dominic Mayers, Michele Mosca, and Harold Ollivier. Self-testing of quantum circuits. In M et al. Bugliesi, editor, *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, number 4052 in Lecture Notes in Computer Science, pages 72–83, 2006. 8, 9

[10] Dominic Mayers and Andrew Yao. Self testing quantum apparatus. *Quantum Information and Computation*, 4(4):273–286, July 2004. 8

[11] Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39:869–877, October 1992. 9

## AUTHOR

Matthew McKague
Centre for Quantum Technologies
National University of Singapore
matthew.mckague@nus.edu.sg