FAST CLASSICAL PERIOD-FINDING OF POLYNOMIALLY-ENTANGLED FUNCTIONS (TOWARDS CLASSICAL SHOR'S)

J. R. "BOB" DOBBS MEMORIAL FAST CLASSICAL PERIOD-FINDING

HOWON LEE
howon@howonlee.com

MARCH 22, 2025

A fully-classical polynomial time and space method of period-finding is given for a set of functions representable polynomially in an idiosyncratic-but-simple iteratively-Kronecker-factored form. Proof is given for the general period-finding case for that set of functions. Some evidence, but not proof, is given that modular exponentiation is within that set of functions, which would entail a fast classical factoring algorithm.

Introduction

> I'm a huge proponent of designing your code around the data, rather than the other way around … I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> — L. Torvalds [1]

[1] Torvalds, L. B. (2006)

Shor's algorithm [2] will be the most important application of quantum computing, whenever we have quantum computers worth a damn. In its most general form it is an algorithm for finding hidden abelian subgroups, which are often used for cryptographical purposes. The most important hidden abelian subgroups in usage are multiplicative factors of integers, followed closely by logarithms of discrete exponents on elliptic curves. [3]

Like substantively all quantum algorithms, Shor's is a hybrid quantum-classical algorithm: the quantum portion serves to

quickly find periods of vast modular exponents. The period-finding, requiring as it does the quantum phase estimation, is usually held to be the important thing, the modular exponentiation held basically as implementation detail. [1]

We give a fully-classical (fully non-quantum) method to find periods of a set of functions quickly (exponentially faster than the naive classical algorithm). The set of functions is defined by polynomial representability as summations of a trivial data structure under some ring.

We give circumstantial evidence, but not proof, that modular exponentiation is within that set. [2]

The method depends deeply upon underlying details of computed functions and therefore is not really compatible with the oracle arguments peeps have for Shor's, but Shor's is not a black box algorithm, unlike Simon's [6]: modular exponentiation is an ordinary computable concrete function in $\mathbb{P}$. [3]

As some random peep eating burritos in San Francisco with no substantive credentials in the field who is substantively motivated by comedy value, our political legitimacy and credibility for claiming any new algorithmic development is poor and we do not really plan to get more credibility. Therefore we have code samples lying around for you to replicate, and the proofs we have of the things we have proofs for are close to trivial.

Everything here is runnable on ordinary consumer classical hardware. To emphasize this fact, we have made all the code copy-and-pastable running Python 3. [4]

Outline

The outline of this document is as follows.

- A data structure to stand in for qubit registers which both *looks like* and *is* trivial,
- A faster FFT on that data structure, corresponding to QFT in the ordinary Shor's case,
- An operation which is vaguely measurement-like and Born's rule-like to index into the amplitude-equivalents on that data structure,

[1] Shor himself did not actually give a gate structure for modular exponentiation (see [4] instead), just noted that it was possible and gave a pseudocode outline of the gates, as modular exponentiation is in $\mathbb{P}$.

[2] This is a different set than functions representable as sets of stabilizer gates compatible with the Gottesman-Knoll theorem [5] (Clifford gates).

[3] It is implementable, it is implemented, peeps actually use it, peeps use it in significant industrial implementation, that sort of thing.

[4] You will need to translate the existing code into a multiprecision numerical library such as [7] if you wish to run things with significant input sizes, however.

- An ansatz on modular operations in general on that data structure, allowing us to give concrete examples of fast period-finding, although currently not on modular exponentiation,
- Worked examples of a trivial and vaguely less trivial period found with such an ansatz.
- A new usage of an obscure hyperoperation which we show can represent an ansatz of modular exponentiation in specific using our data structure, and some notes on that hyperoperation's bad behavior on complex spaces [5]
- And the circumstantial evidence we have amassed that the ansatz can be extended in principle [6], to modular exponentiation in particular. Despite the difficulties with the hyperoperation.

[5] Which is the main thing making us write this whole dealio instead of just showing up with factored large integers.

[6] Given more thinking than a few month's worth.

Data Structure

On the Propbit

A qubit[8] is a linear superposition of two basis states, constrained to be within the Bloch sphere. Usually it is held in quantum computing that the way to represent qubits is within quantum-mechanical devices that form the core portion of a quantum computer, but in simulations it is often common to just represent them as 2-vectors in classical computers, constrained to the Bloch sphere.

This can be done because ordinary vectors are linear, too, even though quantum space is Hilbert space (the physicist's Hilbert space). Qubit registers are, in the pure unentangled state situation, tensor products of these individual qubits. As a basis for actual discrete computations in actual space, the quantum computationalist's Hilbert space is quite different from the actual physicist's Hilbert space inasmuch as it is finite, if exponentially large.

The main way in which our method is pseudo-quantum is in taking this 2-vector semantics of the qubit, which is trivial to put on classical computers, and taking it as far as we can get it, which is surprisingly far. We also take away the Bloch sphere constraint because we work in the extremal, or L_∞ norm, as

opposed to the ordinary quantum L_2 norm, but we suspect that the L_2 norm might be shoved back in there if enough effort is given and might need to be there for a future satisfactory modular exponentiation representation.

This is as opposed to existing quantum simulations [9] on classical computers, which fully expand the quantum amplitudes in order to apply gates which are interpreted as linear operators (matrices), and therefore have to pay the exponential costs involved.

As with other quantum algorithms, Shor's algorithm is cognate to a series of matrix multiplications of complex unitary matrices and then a squaring of the resultant vector to get a probability to sample (Born's rule). The seemingly-turgid notation of quantum physics obscures this fact, but at the same time the turgid notation is necessary because the matrices are absolutely vast, corresponding to finite Hilbert space operators as they do. n qubits indicates a 2^n amplitude vector that we are applying these linear operators on.

The data structure we are using instead of qubits is lists of 2-member lists (or lists of vectors, if you'd like). You do not even have to normalize them, with regards to the stuff we've encountered[7].

```
[[1, 3], [2, 2], [1, 3]]
```

```
[[1, -1], [2, -3 + 2j], [1j, 1]]
```

We have been calling the individual 2-member lists propbits and the lists of lists propbitsets, for propositional bits, because they came out of fiddling around on quantum logic [10] and linear logic [11] we were doing and the contrast possible to ordinary propositional logic. But they really are just lists of lists. We promised that they were actually trivial.

Each individual propbit has two members and corresponds to a qubit, so the whole list corresponds to a register representing a pure unentangled state. This is frankly too trivial to be related nontrivially to tensor networks[12], although we are aware that tensor networks are a thing.

[7] We use the electrical engineering and software notation of j for unit complex and the mathematics and physics notation of i for unit complex, as we will.

Dealing with the propbitsets

The special thing about the propbitsets, and why we are using them to represent qubit registers, is that they, too, can represent the whole vast set of amplitudes without taking too much memory or compute, with the cost that you can have only one amplitude. That cost is paid in ordinary quantum computing, too.

But we must also pay another cost: we need to know an index of the amplitude vector ahead of time. Given that, however, we can just index into the vector at will in a fully classical fashion. One could also think of it as being forced to be computationally lazy, where the evaluations being avoided comprise the entire rest of the amplitude vector.

We are assuming initially pure states, so to get the index, we take our index into the amplitude, turn it into a binary representation, index into the individual propbits in the propbitset with the individual members of the binary representation, and then multiply all the members of the propbits together.

```
import numpy as np

def idx_fn(pbset, idx):
    members = []
    idx_repr = list(map(int, np.binary_repr(idx, width=len(pbset))))
    for propbit_idx, propbit_val in enumerate(idx_repr):
        members.append(pbset[propbit_idx][propbit_val])
    return np.prod(members)
```

```
>> example_pbs = [[1, 2], [2, -3], [5, 7]]
>> idx_fn(example_pbs, 6)
-30
```

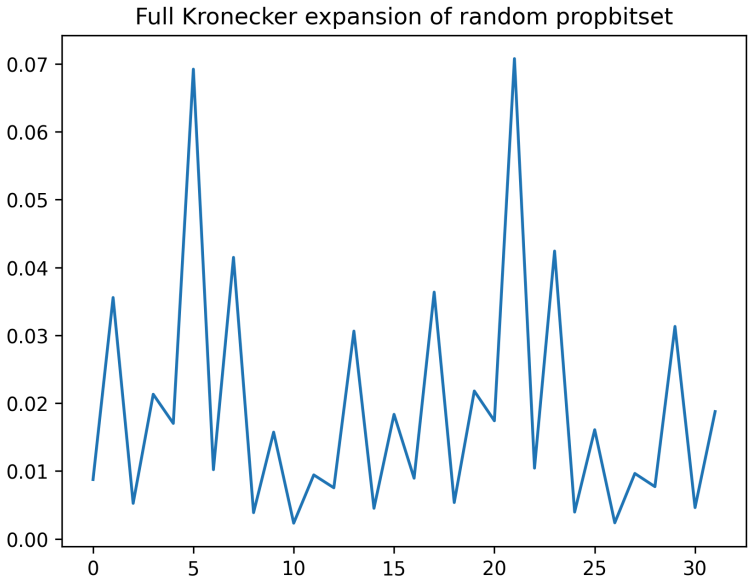
The full expansion (which is not practicable with a nontrivial number of propbits, but we can do it for our own sakes or for making graphs) of these propbitsets with these individual indexes for each individual index of the 2^n amplitude vector looks like taking the iterated Kronecker product of them, so the Kronecker product [13] is our analog of the tensor product here.

Often, peeps declare that the Kronecker product is a tensor product. Also often, peeps also give definitions for the tensor product which are not perfectly compatible with the Kronecker product, usually by being strict generalizations. The Kronecker

product is unambiguously a concretion, and therefore as we are doing actual computations we think with Kronecker products in this case.

```
import functools
import numpy as np

def all_amplitudes(pbset):
  return functools.reduce(np.kron, pbset)
```

```
>> example_pbs = [[1, 2], [2, -3], [5, 7]]
# These do not correspond to the figure, because the figure is actually random
>> all_amplitudes(example_pbs)
array([ 10,  14, -15, -21,  20,  28, -30, -42])
```



Note that we could also have lists of 2x2 matrices, and these would be indexable within the $2^n \times 2^n$ matrix but also Kronecker-factorable. [8]

Recall the mixed-product property of Kronecker products ($\otimes$ as Kronecker product):

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD). \tag{1}$$

Therefore, if you wish to multiply a propbitset with a matrix which can be factored into itself a propbitset, one can factorize everything down. A propbitset representing a set of 2^n amplitudes can thus be multiplied with, effectively, a $2^n \times 2^n$ matrix in $O(n)$ time, given that factorization condition.

[8] Play around a bit with these and you will notice that they can be used to model some fractals. Probably infinite sets of propbits with noninteger Besicovich dimensions could be found, although we do not know of a use for that.

```
def matrix_product(pbset, matrix_pbset):
    res_pbset = []
    for pb, matrix_pb in zip(pbset, matrix_pbset):
        res_pbset.append(pb @ matrix_pb)
    # result is a pbset
    return res_pbset
```

```
>> example_pbs = [[1, 2], [2, -3]]
>> example_matrix_pbs = [
  np.array([[6, 1], [-1, 3]]),
  np.array([[2, 3], [7, -3]])]
>> matrix_product(example_pbs,
example_matrix_pbs)
[array([4, 7]), array([-17,  15])]
```

Proof we correspond to the full matrix product: Base case is trivial. Induct with mixed product property.

Given semantics for interleaving of matrix multiplications, it becomes also straightforward to deduce a method of taking the inner product of two propbitsets.

```
import numpy as np

def inner_product(fst_pbset, snd_pbset):
    res = 1. + 0j
    for fst_pb, snd_pb in zip(fst_pbset, snd_pbset):
        res *= np.dot(fst_pb, snd_pb)
    return res
```

```
>> example_pbs_1 = [[1, 2j], [2, -3]]
>> example_pbs_2 = [[3, -2], [11j,
4]]
>> inner_product(example_pbs_1,
example_pbs_2)
(52+114j)
```

Proof we correspond to the full dot product: Base case is trivial, induct with mixed product property. Actual concrete implementation is of dot product only, but generalization to inner product is also straightforward.

Given a propbitset represents a 2^n vector with n propbits, the inner product, which is obviously $O(n)$, comprises an inner product of a 2^n vector by another 2^n vector in $O(n)$ time. But with many restrictions, again, on the form of the vector.

If one notes that to reverse the amplitude vector corresponding to the propbitset, one can simply reverse each individual propbit in the propbitset, getting individual amplitudes of a cyclic convolution also becomes straightforward, as the cyclic convolution is itself an inner product.

Currently the best way we've found to deal with inner products of entangled systems is by taking them pairwise as pairs of pure states.

Non-considerations

Because Shor's is not a general computation, only a Hadamard product, conditioned modular exponentiation, quantum Fourier transform then a measurement in Hilbert space, we do not lay out a universal gateset or generally treat on quantum logic gates, instead working more directly with what entangled sums of propbitsets we can get. We do not have uncomputation, and our method of both getting at functional superposition and doing phase kickback is much more direct.

This is also why we do not use quantum notation, because ordinary linear-algebraic notation and concrete software is enough for us.

Algebraic considerations

Recall that a semigroup is a set endowed with an associative binary operation. Recall that an algebraic ring is a set endowed with two binary operations, called addition and multiplication.

For our purposes for the rest of this document addition and multiplication in a ring will be called, respectively, the lesser and greater operations, because we will have a ring with ordinary multiplication as the *lesser* operation.

The lesser and greater operation in a ring are such that the lesser operation commutes and associates and has identity, and such that the greater operation associates, has an identity, and distributes over the lesser operation.

The avoiding-computation semantics of the propbitsets require only a semigroup to work. Associativity is required because of the multi-propbit nature of the propbitset.

However, the factored fast matrix multiplication and the fast inner product require, respectively, a ring and a ring with an inner product space defined over it. Taking advantage of this fact is our main attempted method of sidestepping complicated entanglement, but currently we have not found it to be sufficient as of yet.

Let us give some examples of non-multiplication semigroups amenable to the propbitset treatment. Try the sum operation:

```
import numpy as np

def idx_fn(pbset, idx):
    members = []
    idx_repr = list(map(int, np.binary_repr(idx, width=len(pbset))))
    for propbit_idx, propbit_val in enumerate(idx_repr):
        members.append(pbset[propbit_idx][propbit_val])
    return np.sum(members)
```

```
>> example_propbitset = [[0, 2], [4,
4], [2, 9j]]
>> idx_fn(example_propbitset, 3)
(4+9j)
```

and the XOR operation.

```
import numpy as np
import functools

def idx_fn(pbset, idx):
    members = []
    idx_repr = list(map(int, np.binary_repr(idx, width=len(pbset))))
    for propbit_idx, propbit_val in enumerate(idx_repr):
        members.append(pbset[propbit_idx][propbit_val])
    return functools.reduce(np.logical_xor, members)
```

```
>> example_propbitset = [
  [True, False],
  [False, False],
  [False, True]
]
>> idx_fn(example_propbitset, 3)
False
```

As one can see, the simplicity of propbitsets means that it is also simple to get the results of the resultant vector on the semigroup operations on propbitsets.

Given that one could use any semigroup, one would think that one particular semigroup of interest to use would be modular multiplication with the wanted modulus to factor. Then, it seems straightforward to represent the modular exponentation all at once in a single propbitset, then take the psedoquantum Fourier transform in the way we will detail later and measure in the way we will also detail later and the pseudoquantum Shor's seems straightforward after that.

In fact we have not found this so useful. This is because, although the modular multiplication gives us a ring along with modular addition, the Fourier transform over that ring (the number-theoretic transform [14], or computationally-equivalently the finite-field Fourier transform) seems incompatible with our way of getting measurements, because our way of getting measurements depends upon the duality of the period of the periodic function and the period of the Fourier transform of

that function in a way that seems incompatible with the number-theoretic transform.

Perhaps it is not so, and yet another method of getting measurements is possible done with the number-theoretic transform, and also that a period-finding method can be found with the extremal number theoretic transform, whereupon much of the following document sort of dissolves. But we have not found such a method, either of getting measurements or of finding periods.

Entanglement

The notion of entanglement translates straightforwardly to propbitsets by addition of the propbitsets, or the lesser operation in the ring the propbitset is defined in. The summation must be done lazily (by this we mean computational laziness, in the Haskell style), after evaluation of the propbitset (after indexing into the amplitude vector), as far as we can tell.

The Bell state in propbitsets goes like so:

```
[[1, 0], [1, 0]] + [[0, 1], [0, 1]]
```

Extension to Greenberger-Horne-Zeilinger state is straightforward. For state of n qubits, the GHZ state in propbitsets goes like:

```
[[1, 0] for idx in range(n)] + [[0, 1] for idx in range(n]]
```

You can normalize if you would like, but we do not care that much.

Note that propbitsets defined with regards to a ring are simply entangled with the lesser operation in the ring. Consider the Boolean ring with lesser operation XOR and greater operation AND. The Bell and GHZ state therefore translates directly in to the Boolean ring case, to

```
[[True, False] for idx in range(n)]
XOR
[[False, True] for idx in range(n)]
# If you want to actually run this,
# the XOR operator in Python is actually `^`.
# You must also make your own provisions for laziness.
```

Yet Another Entanglement Measure

There are many quantities which can be theorized pertaining to entanglement, because entanglement is a quantum correlation measure and there are many measures of correlation [15]. Most are operational with respect to physical information theory, not direct computation. Extremal properties and distillation are also possible, but not helpful for our purposes, which are solely computational and purely classical in nature.

We propose a measure of a function with respect to the fully-classical entanglement-like property of propbitsets for its minimal formulation in terms of sums of propbitsets, where entanglement is imagined as a resource taken up computationally, like time, space or entropy.

The measure would consist of the cardinality of propbitsets needed to sum to represent the superposition of the amplitude vector. For example, the Bell state takes 2 propbitsets to represent, so our putative measure of classical entanglement complexity would measure 2 in that case. This is quite analogous to the situation where one can write $O(n \log n)$ time complexity sorts or $O(n^2)$ time complexity sorts, the complexity is with respect to the implementation.

Regrettably for our purposes, there does exist a measure called entanglement complexity [16], and it is not this measure. We have been calling the quantity yaem (Yet Another Entanglement Measure) to be sufficiently idiosyncratic, in programmer style.

Like other computational resources, we suggest using Bachmann-Landau notation to denote the consumption with respect to input sizes this resource, where input sizes will mostly be number of propbits in the propbitset.

Some example functions with yaems

For the trivial function $f(x) = x$, we give an implementation for amplitude vectors corresponding to n propbits (so 2^n size amplitude vectors) in $O(n)$ yaem.

```
import numpy as np

def linear_idx_fn(num_pbsets):
    pbs = []
    for curr_pbs_idx in range(num_pbsets):
        curr_pbs = [[1, 1] for _ in range(num_pbsets)]
        curr_pbs[curr_pbs_idx] = [0, 2 ** (num_pbsets - curr_pbs_idx -
1)]
        pbs.append(curr_pbs)

    def idx_fn(idx):
        final_res = 0
        idx_repr = list(map(int, np.binary_repr(idx, width=num_pbsets)))
        for curr_pbset in pbs:
            curr_member = 1
            for propbit_idx, propbit_val in enumerate(idx_repr):
                curr_member *= curr_pbset[propbit_idx][propbit_val]
            final_res += curr_member
        return final_res
    return idx_fn
```

Derive by noting that each bit in the overall linear sequence can correspond to a propbitset. Each curr_pbs is one propbitset, so n yaems.

```
def full_fn_expand(idx_fn,
num_propbits):
    return np.array(
      [idx_fn(idx)
      for idx in
      range(int(2 ** num_propbits))]
    )

>> curr_idx_fn = linear_idx_fn(4)
>> full_fn_expand(curr_idx_fn, 4)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10, 11, 12, 13, 14, 15])
>> curr_idx_fn(6)
6
```

The mildly less trivial function $f(x) = x^2$. We give something in $O(n^2)$ yaems. We make no claims as to these being tight bounds. To say that sums of propbitsets are poorly determined representations is an understatement in and of itself.

```
import numpy as np

def square_idx_fn(num_pbsets):
    pbs = []
    for curr_pbs_idx_fst in range(num_pbsets):
        for curr_pbs_idx_snd in range(num_pbsets):
            curr_pbs = [[1, 1] for _ in range(num_pbsets)]
            curr_pbs[curr_pbs_idx_fst] = [0, 1]
            curr_pbs[curr_pbs_idx_snd] = [0, 2 ** ((2 * (num_pbsets - 1)) -
curr_pbs_idx_fst - curr_pbs_idx_snd)]
            pbs.append(curr_pbs)

    def idx_fn(idx):
        final_res = 0
        idx_repr = list(map(int, np.binary_repr(idx, width=num_pbsets)))
        for curr_pbset in pbs:
            curr_member = 1
            for propbit_idx, propbit_val in enumerate(idx_repr):
                curr_member *= curr_pbset[propbit_idx][propbit_val]
            final_res += curr_member
        return final_res
    return idx_fn
```

Outer and inner loops of length n each in the propbitset generation loop, so $O(n^2)$ yaem. Derive by recalling that $n^2 = \sum_{k=1}^{n}(2k-1)$ and applying the binary trick we had for the linear case.

```
def full_fn_expand(idx_fn,
num_propbits):
    return np.array(
      [idx_fn(idx)
      for idx in
      range(int(2 ** num_propbits))]
    )

>> curr_idx_fn = square_idx_fn(4)
>> full_fn_expand(curr_idx_fn, 4)
array([  0,   1,   4,   9,  16,  25,  36,
49,  64,  81, 100, 121, 144, 169, 196,
225])
>> curr_idx_fn(6)
36
```

Real exponent. This can be done in 1 propbitset, with use of the doubly exponential series.

```
import numpy as np

def real_exponent_fn(base, num_pbsets):
    pbs = []
    for member in range(num_pbsets):
        curr_member = base ** (2 ** (num_pbsets - member - 1))
        pbs.append([1, curr_member])

    def idx_fn(idx):
        res = 1
        idx_repr = list(map(int, np.binary_repr(idx, width=num_pbsets)))
        for propbit_idx, propbit_val in enumerate(idx_repr):
            res *= pbs[propbit_idx][propbit_val]
        return res
    return idx_fn
```

```
def full_fn_expand(idx_fn,
num_propbits):
    return np.array(
      [idx_fn(idx)
      for idx in
      range(int(2 ** num_propbits))]
    )

>> curr_idx_fn =
real_exponent_fn(2, 4)
>> full_fn_expand(curr_idx_fn, 4)
array([   1,    2,    4,    8,   16,   32,
64,  128,  256,
        512, 1024, 2048, 4096, 8192,
16384, 32768])
>> curr_idx_fn(6)
64
```

Yaem measures should be with respect to the semigroup operation on the propbitset. Real exponent can be done with respect to multiplication quickly, but not with addition, for example.

Difficulties with modular functions

If we get into the mindset of thinking of the classical entanglement as a resource like time, space or entropy, obviously we can classify functions as having polynomial or exponential yaem complexity to represent them in classical superposition, just as we can classify them as having polynomial or exponential time complexity, space complexity, etc.

Exponential yaem is useless for our purposes. Anyone can factor as much as they would like, given exponential time.

Regrettably we have not found a way to represent modular functions in polynomial yaem straightforwardly in their original form. See the section on the complex ansatz and the commutative hyperoperation for our putative (but incomplete) workaround.

Fast Pseudoquantum Fourier Transform

Recall the discrete Fourier transform [17], defined as transforming a vector x to the frequency vector X:

$X_k = \sum_{n=0}^{N-1} x_n e^{-2i\pi \frac{k}{N} n}$

Note that each member of the frequency-valued FFT result is an inner product itself, of the data with respect to a finite geometric sequence in the complex exponentials. The sequence is different for each member of the result Fourier-transformed vector. Done with respect to quantum amplitudes, the discrete Fourier transform is called the quantum Fourier transform [18].

Note that QFT has been done quickly in classical simulation before without propbit semantics [19], so the novelty remains the measurement-like process we do and the period-finding we get from it. However, the propbit semantics should also be much easier to go find alternate uses for.

Consider realizations from a geometric sequence.

$a, ar, ar^2, ar^3, ar^4, ...$

Given a finite set of realizations from geometric sequence, the realizations being of length 2^n, it is uncomplicated to Kronecker-factorize it out into a propbitset of n propbits. Given is an example for $2^n = 16, n = 4$.

$[[a, ar^8], [1, r^4], [1, r^2], [1, r]]$

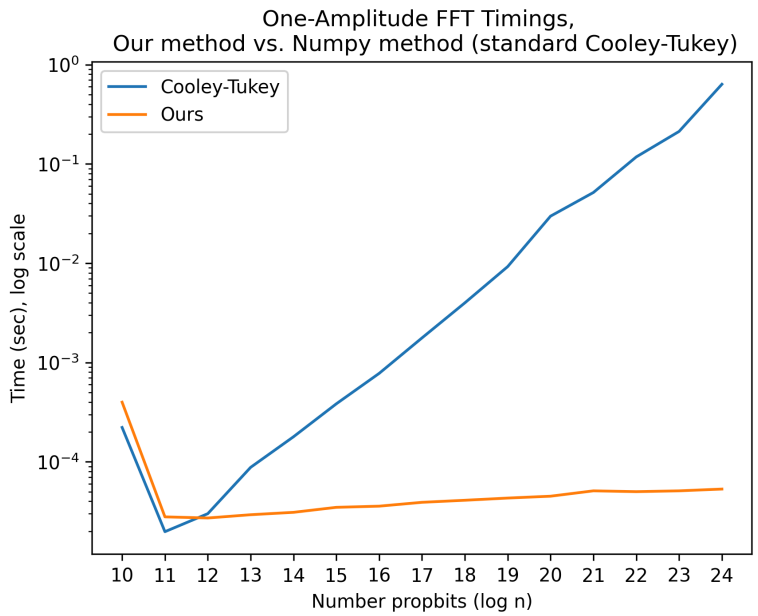
In the general case,

$\left[\left[a, ar^{2^{n-1}}\right], \left[1, r^{2^{n-2}}\right], ...[1, r]\right]$

Recall DFT for one index is an inner product with respect to a complex-valued geometric sequence. Therefore, if the data is in the form of a propbitset, the discrete Fourier transform of it for a specific index is the inner product of the initial propbitset with this other propbitset. We have shown previously that this can be done exponentially faster than ordinary inner products.

Here is an implementation of this faster Fourier transform, for propbitsets only.

```
import numpy as np

def unity_root(m):
    """
    2 ** mth root of unity, so we can just put in num_propbits as m
    """
    return np.exp((2j * np.pi) / (2 ** m))

def fft_propbits(num_propbits, idx):
    res = [np.array([0, 0]) for _ in range(num_propbits)]
    for propbit_idx in range(num_propbits):
        exp = -(2 ** (num_propbits - propbit_idx - 1)) * idx # un-negate
for inverse transform
        res[propbit_idx] = np.array([1 + 0j, unity_root(num_propbits) **
exp])
    return res

# Inner product reproduced here for your convenience
def inner_product(fst_pbset, snd_pbset):
    res = 1
    for fst_pb, snd_pb in zip(fst_pbset, snd_pbset):
        res *= np.dot(fst_pb, snd_pb)
    return res
```




```
>> example_propbitset = [[0, 0.5],
[0.4, 0.4], [0.2, 0.9j]]
# Examine ordinary fft of the fully
expanded result
>> import functools;
np.fft.fft(functools.reduce(np.kron,
example_propbitset))
array([ 0.08      +0.36j,
-0.29455844+0.04j,  0.        +0.j  ,
       -0.29455844-0.04j,  0.08
-0.36j,  0.21455844+0.04j,
       0.        +0.j  ,  0.21455844-0.04j])
# Now do it our way
>> fft_idx_propbits =
functools.partial(fft_propbits, 3)
>>
inner_product(example_propbitset,
fft_idx_propbits(5))
(0.21455844122715712+0.040000000000000785j)
>>
[np.round(inner_product(example_propbitset,
fft_idx_propbits(idx)), 9) for idx in
range(8)] # round to make the
result less ugly
[(0.08+0.36j), (-0.294558441+0.04j),
-0j, (-0.294558441-0.04j), (0.08-0.36j),
(0.214558441+0.04j), (-0+0j),
(0.214558441-0.04j)]
```

Fourier transform remains linear [9], so the entangled case with entanglement by summation is straightforward, just add the individual propbitset results. Therefore, the pseudoquantum Fourier transform of a function, given the index, should be the same yaem as the function itself. At least of a function represented as sums of the ordinary multiplication semigroup.

Many other discrete integral transforms are feasible. One in particular is very easy. The Walsh-Hadamard transform [20], or Hadamard transform, with Hadamard matrix done via the usual Sylvester construction can be Kronecker-factorized very easily because the construction is a Kronecker product in the first place.

That is, recall that the Sylvester construction of the Hadamard matrix, which is the matrix corresponding to the Hadamard operator, of size $2^n \times 2^n$ is

$$\otimes_n \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

where $\otimes$ is Kronecker product. One can then normalize to unitarity if one wishes. Therefore we can give very succinctly the Hadamard transform as a matrix-vector multiplication of propbitsets.

[9] That is, in the additivity-and-homogeneity sense, although it is also incidentally linear in the number of propbits in the Bachmann-Landau sense

```
import numpy as np

def hadamard_propbits(num_propbits):
    """Unnormalized, normalization is straightforward"""
    return [np.array([[1, 1], [1, -1]]) for _ in range(num_propbits)]

# Matrix product reproduced here for your convenience
def matrix_product(pbset, matrix_pbset):
    res_pbset = []
    for pb, matrix_pb in zip(pbset, matrix_pbset):
        res_pbset.append(pb @ matrix_pb)
    return res_pbset
```

```
>> example_propbitset = [[0, 0.5],
[0.4, 0.4], [0.2, 0.9j]]
>> import scipy.linalg as sci_lin
>> import functools
# Examine ordinary hadamard of
the fully expanded result. Usually
folks do fast Hadamard transform
instead.
>> functools.reduce(np.kron,
example_propbitset) @
sci_lin.hadamard(8)
array([ 0.08+0.36j,  0.08-0.36j,  0.
+0.j  ,  0.  +0.j  , -0.08-0.36j,
       -0.08+0.36j,  0.  +0.j  ,  0.  +0.j  ])
>> fast_had_res =
matrix_product(example_propbitset,
hadamard_propbits(3))
>> fast_had_res
[array([ 0.5, -0.5]), array([0.8, 0. ]),
array([0.2+0.9j, 0.2-0.9j])]
>> functools.reduce(np.kron,
fast_had_res)
array([ 0.08+0.36j,  0.08-0.36j,  0.
+0.j  ,  0.  +0.j  , -0.08-0.36j,
       -0.08+0.36j, -0.  +0.j  ,  0.  +0.j  ])
```

Ordinarily, the treatment of quantum Fourier transform is done by a Hadamard transform and a series of twiddles. This can also be done in the propbitset case.

```
import numpy as np
import copy

def unity_root(m):
    """ 2 ** mth root of unity, so we can just put in num propbits as m
"""
    return np.exp((2j * np.pi) / (2 ** m))

def pb_twiddle_fft(propbits, idx):
    res = 1. + 0j
    num_propbits = len(propbits)
    idx_repr = list(map(int, np.binary_repr(idx, width=num_propbits)))
    # bit reversal
    new_idx_repr = list(reversed(idx_repr))
    # do not mutate passed in propbits
    curr_propbits = copy.deepcopy(propbits)
    for propbit_idx, propbit_val in enumerate(new_idx_repr):
        curr_propbit = curr_propbits[propbit_idx]
        # Hadamard transform
        had_res = [
                curr_propbit[0] + curr_propbit[1],
                curr_propbit[0] - curr_propbit[1]
                ]
        res *= had_res[propbit_val]
        # series of twiddles
        if propbit_val == 1:
            for offset, other_pb_idx in enumerate(range(propbit_idx + 1,
num_propbits)):
                curr_propbits[other_pb_idx][1] *= unity_root(offset + 2)
    return res
```

See [18] for a treatment of the ordinary QFT in quantum computers done via Hadamard transform and a series of twiddles.

Note that, even if one were to ignore the period-finding semantics possible with this pseudoquantum FFT, one cannot so much as throw a stone in all of applied mathematics without hitting some profitable use for the FFT. Therefore, if one can coerce data into a Kronecker-factored form (into a propbitset form), knowing that one amplitude of the FFT can be taken in

```
>> example_propbitset = [[0, 0.5],
[0.4, 0.4], [0.2, 0.9j]]
>>
[np.round(pb_twiddle_fft(example_propbitset,
idx), 9) for idx in range(8)]
[(0.08+0.36j), (0.214558441-0.04j),
(-0+0j), (0.214558441+0.04j),
(0.08-0.36j), (-0.294558441-0.04j), 0j,
(-0.294558441+0.04j)]
```

time linear in the number of propbits (and therefore exponentially faster than the whole amplitude vector FFT, if one doesn't need the whole amplitude vector), is almost certainly still valuable knowledge outside of hidden-abelian-subgroup finding.

There is a material literature on the ordinary sparse FFT [21]. Almost all are materially more flexible with regards to arbitrary data, but they are not so well-suited for our specific purpose of getting that single amplitude. Many are sampling-based in nature, whereas this FFT is not actually probabilistic and we add the probabilistic semantics in another portion of the procedure.

Extrema search as L_∞ sampling

Quantum physics is L_2 norm physics. After the linear operations which constitute all quantum operations of any note, quantum computing requires a measurement step consisting of application of Born's rule in order to actually do computations, and Born's rule is a translation from the amplitudinal space to the probabilistic by way of the L_2 norm. Also see [22].

We use extremal values on the amplitude vector, more reminiscent of the L_∞ norm instead, in order to translate from the amplitude space to actual computations. This is because there is a straightforward way to get local extrema quickly, by having some state for indices and by using a divide-and-conquer method.

Recall the L_∞ norm, sometimes called the Chebyshev norm.

$L_\infty(x, y) = \max_i |x_i - y_i|$

This requires global maxima on absolute differences. Absoluteness of differences can be had after the fact.

Norms are dual to probability-like measures. L_1 corresponds to ordinary probability. L_2 corresponds to amplitudes taken by Born's rule as in ordinary quantum mechanics. We have been thinking of the local maxima we have been getting to get the periods of things as a sort of diseased deterministic probability measure, because the extrema search would assign entire basins deterministically but chaotically to extrema.

If we see Born's rule as sampling on the L_2 norm, we can also see getting local maxima on the L_∞ norm as a sort of sampling. This sort of sampling, importantly, seems more viable to us with classical computation. Recall that a difficulty of our method of getting amplitudes in the large amplitude vector is the necessity of having an index at every time. If we 'sample from the L_∞ norm', with ordinary-for-quantum-algorithms smoothness guarantees, then we can use a stateful local search algorithm to keep an index as state. This is therefore a cogent reason to go to this strange norm.

There are almost certainly guarantees needed on the number of critical points in the frequency space which we need for this to work, because something which is fractal in the frequency space will not have useful extrema for our purposes. However, we just care about the spectra of modular functions and other known-periodic functions, which are well behaved in the frequency space.

Algorithm

We can find the actual local maximum with a divide-and-conquer algorithm [23]. We cannot find a non-didactic citation for this algorithm, and peeps seem to not cite it in non-didactic situations, perhaps for its triviality, as basically gussied-up binary search. For example, Scipy [24] does not cite its materially variant implementation at all.

Usually the didactic presentation is given with an entire instantiated array in which a peak must be found quickly, but it works fine with our lazy non-instantiated vector of amplitudes.

```
import numpy as np

# pass in a curried idx_fn
def peak_find(idx_fn, start, end, mult=5):
    curr_start, curr_end = start, end
    cutoff = int(np.log(end - start) * mult)
    for _ in range(cutoff):
        middle = (curr_start + curr_end) // 2
        if np.abs(idx_fn(middle)) < np.abs(idx_fn(middle - 1)):
            curr_end = middle - 1
        elif np.abs(idx_fn(middle)) < np.abs(idx_fn(middle + 1)):
            curr_start = middle + 1
```

```
    else:
        return middle
return None
```

See [23] for correctness. Speed is found via master method to be $O(f(a)\log a)$, where $f(a)$ is the time to execute the idx_fn for an individual index (which should be done with respect to the propbitsets, so polylog in a) and a is cardinality of amplitudes if they were to be fully instantiated, which we must avoid doing. Space is $O(g(a))$, where $g(a)$ is the amount of space for the idx_fn, which should also be polylog with respect to the whole size of the amplitudes.

Given a peak value for the Fourier transform of a function, the rest of Shor's algorithm is entirely classical in nature.

Correctness of Peak-Finding to Period-Finding

Recall that the Fourier transform of a fully-defined periodic Fourier series has peaks only at integer multiples of the frequency, from which it is trivial to get the period.

Consider some function $x(t)$, period r, within our domain from 0 to $Q-1$, construed as a sum of Dirac combs:

$$x(t) = \sum_{k=0}^{L} d_k \delta(t - kr) \tag{2}$$

where L is $\left\lfloor \frac{Q}{r} \right\rfloor$.

with Fourier coefficients c_k:

$$X(t) = \sum_{k=0}^{L} c_k e^{jk\omega_0 t} \tag{3}$$

But construe this as a geometric sum, and we see that we get constructive interference when $\frac{Q}{r}$ is close to integer and destructive when not, just as in ordinary Shor's.

If $r \mid Q$ (coset of the subgroup, specifically), where Q is our amplitude cardinality, this is exact constructive and destructive interference.

Finding maximum means we find the peak of one the Dirac spikes, which is now a good approximation for $\frac{Q}{r}$, as in Shor's. If $r \mid Q$ perfectly we cannot actually do this, because the divide-and-conquer algorithm depends upon slope information,

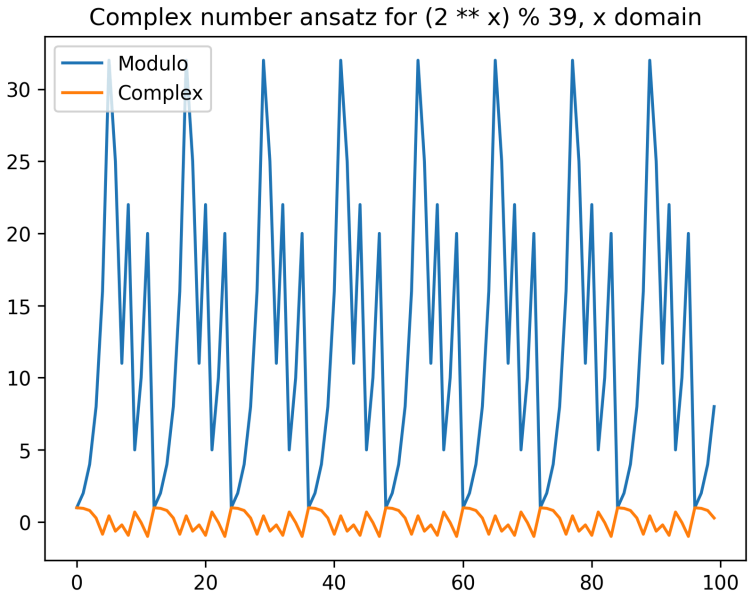
but if r does not divide into Q perfectly, note that there will be slope information.

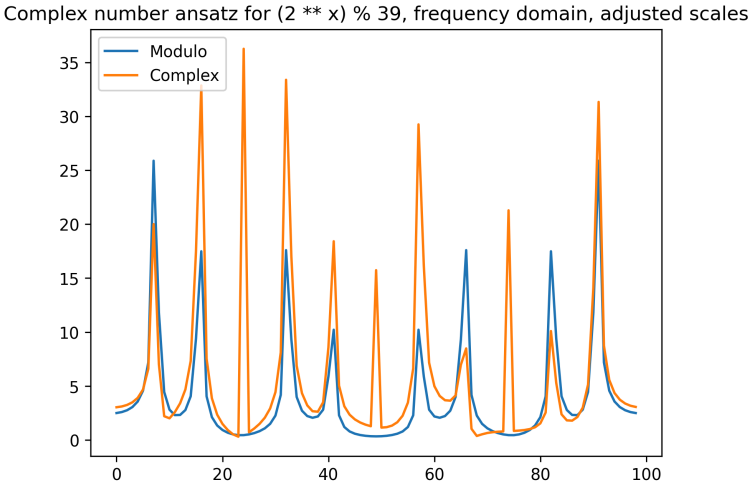
The need for getting to the actual ratio after getting the approximation, and the solution for same, continued fractions, is substantively identical to ordinary Shor's[2].

Complex-Number Putative Ansatz

We now give a putative ansatz compatible with the propbitsets for dealing with the open-ended entangling on functions taken modulo n. It seems to us that the ansatz is only really good for finding periods of $f(x) \bmod n$, but that is what we are doing, so here we go. We call this a putative ansatz because the actual core assumption of the functional form is not yet validated.

Putative ansatz: Consider a function $f(x) \bmod n$. We say as approximation that the period of the function is roughly equivalent to the period of $e^{i\frac{f(x)}{n}}$.





As the complex exponential of a function is often far easier to cram into an exponential sequence formula or sums thereof than the modulo of the function, and individual exponential sequences are always representable with one unentangled propbitset, we think of this as more compatible with the propbitsets.

Given a representation of the function in a more unentangled manner, in order to get the period in actuality we have to get the fast classical QFT working on the representation. As you will see, this is actually the main obstacle, if we use the ansatz.

Note the similarity in the pure unentangled state of the whole apparatus to addition by QFT [25].

Worked Example

An implementation of that remaining portion is provided for convenience's sake. [26] is a library for continued fractions, used here.

```
import functools
import time
import math
import collections
import numpy as np
import numpy.random as npr
import fractions
import contfrac  # get this library from pip or github - https://github.
```

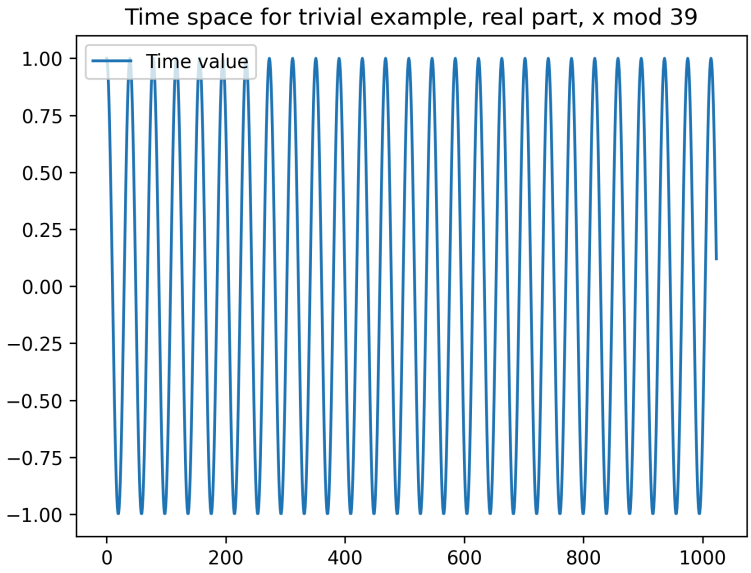
com/TheMatjaz/contfrac - cited above

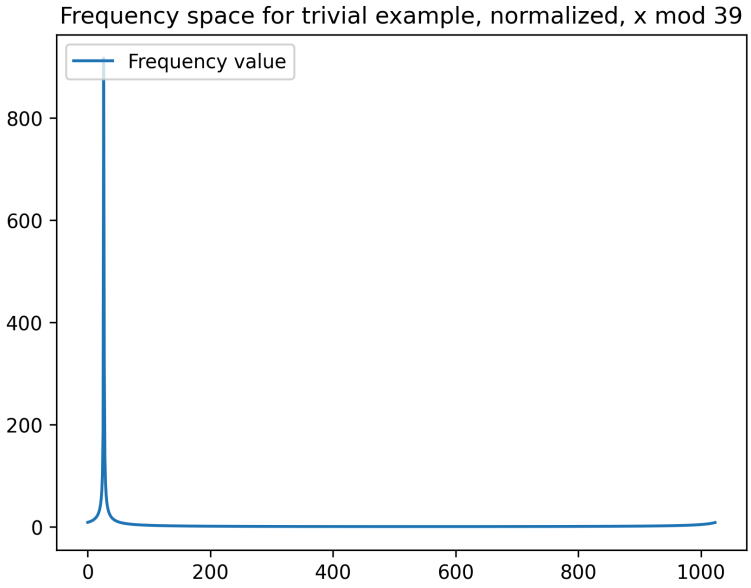
```
def peak_find(idx_fn, start, end, mult=5):
    curr_start, curr_end = start, end
    cutoff = int(np.log(end - start) * mult)
    for _ in range(cutoff):
        middle = (curr_start + curr_end) // 2
        if np.abs(idx_fn(middle)) < np.abs(idx_fn(middle - 1)):
            curr_end = middle - 1
        elif np.abs(idx_fn(middle)) < np.abs(idx_fn(middle + 1)):
            curr_start = middle + 1
        else:
            return middle
    return None


def peaklist(idx_fn, total_end, num_peaks):
    peaks = list()
    while len(peaks) < num_peaks:
        curr_start, curr_end = sorted(list(npr.randint(0, total_end,
size=2)))
        # This should really be not a thing with nontrivial numbers of
propbits,
        # but it was annoying me testing trivial numbers
        while curr_start == curr_end:
            curr_start, curr_end = sorted(list(npr.randint(0, total_end,
size=2)))
        next_peak = peak_find(idx_fn, curr_start, curr_end)
        if next_peak:
            peaks.append(next_peak)
    return peaks


def contfrac_denom(peak, num_propbits):
    num_members = 2**num_propbits
    value = (peak, num_members)
    convs = list(contfrac.convergents(value))
    for convergent in convs:
        subtrahend = np.abs(
            fractions.Fraction(*value) - fractions.Fraction(*convergent)
        )
        if subtrahend < (fractions.Fraction(1, 2 * num_members)):
            return convergent
```

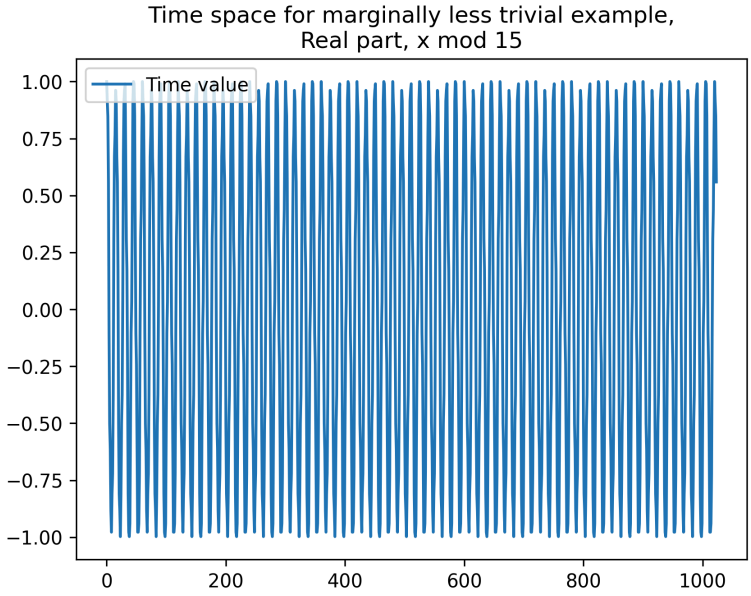
```
    return None


def test_period(mod, num_propbits, idx_fn):
    assert mod % 2 == 1, "Come on..."
    num_peaks = 3
    start_time = time.time()

    num_iters = 100
    period_set = collections.Counter()
    for i in range(num_iters):
        total_end = int(2**num_propbits) - 1
        peaks = peaklist(idx_fn, total_end, num_peaks)
        contfracs = list(map(lambda x: contfrac_denom(x, num_propbits),
peaks))
        print([2 ** num_propbits / peak for peak in peaks])
        contfracs = list(filter(lambda x: x, contfracs))
        lcm = functools.reduce(np.lcm, map(lambda x: x[1], contfracs), 1)
        period = lcm

        print(
            i,
            start_time - time.time(),
            peaks,
            contfracs,
            lcm,
            lcm / float(mod),
        )
        period_set[period] += 1
    print("nontrivial period set: ", period_set)


def unity_root(m):
    """2 ** mth root of unity, so we can just put in num propbits as m"""
    return np.exp((2j * np.pi) / (2**m))


def fft_propbits(num_propbits, idx):
    res = [np.array([0, 0]) for _ in range(num_propbits)]
    for propbit_idx in range(num_propbits):
        exp = -(2 ** (num_propbits - propbit_idx - 1)) * idx
        res[propbit_idx] = np.array([1 + 0j, unity_root(num_propbits) **
exp])
    return res
```

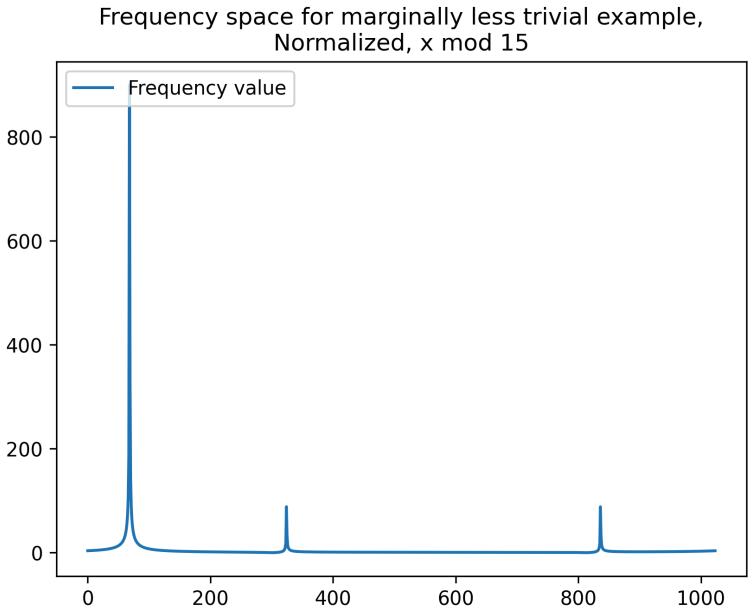
```
# Note the off-by-ones on these


def idx_fn_trivial_example(idx, mod, num_propbits):
    propbits = [
        np.array([1, np.exp(2j * np.pi * (2** (i - 1)) / mod)]) for i in
range(num_propbits, -1, -1)
    ]
    curr_fft_propbits = fft_propbits(num_propbits, idx)
    res = 1
    for propbit_idx in range(num_propbits):
        res *= propbits[propbit_idx].dot(curr_fft_propbits[propbit_idx])
    return res



def idx_fn_low_yaem_entangled_example(idx, mod1, mod2,
num_propbits):
    propbits_1 = [[1, np.exp(2j * np.pi * (2 ** (i - 1)) / mod1)] for i in
range(num_propbits, -1, -1)]
    propbits_1 = list(map(np.array, propbits_1))
    propbits_2 = [[1, np.exp(2j * np.pi * (2 ** (i - 1)) / mod2)] for i in
range(num_propbits, -1, -1)]
    propbits_2 = list(map(np.array, propbits_2))
    curr_fft_propbits = fft_propbits(num_propbits, idx)
    res_1, res_2 = 1, 1
    for propbit_idx in range(num_propbits):
        res_1 *=
propbits_1[propbit_idx].dot(curr_fft_propbits[propbit_idx])
        res_2 *=
propbits_2[propbit_idx].dot(curr_fft_propbits[propbit_idx])
    return res_1 + res_2


def idx_fn_marginally_less_trivial_example(idx, mod, num_propbits):
    propbits = [[1, np.exp(2j * np.pi * (2 ** (i - 1)) / mod)] for i in
range(num_propbits, -1, -1)]
    propbits[8] = [1, np.exp(2j * np.pi * (2 ** (num_propbits - 8 - 1)) /
(mod * 3))]
    propbits = list(map(np.array, propbits))
    curr_fft_propbits = fft_propbits(num_propbits, idx)
    res = 1
    for propbit_idx in range(num_propbits):
```

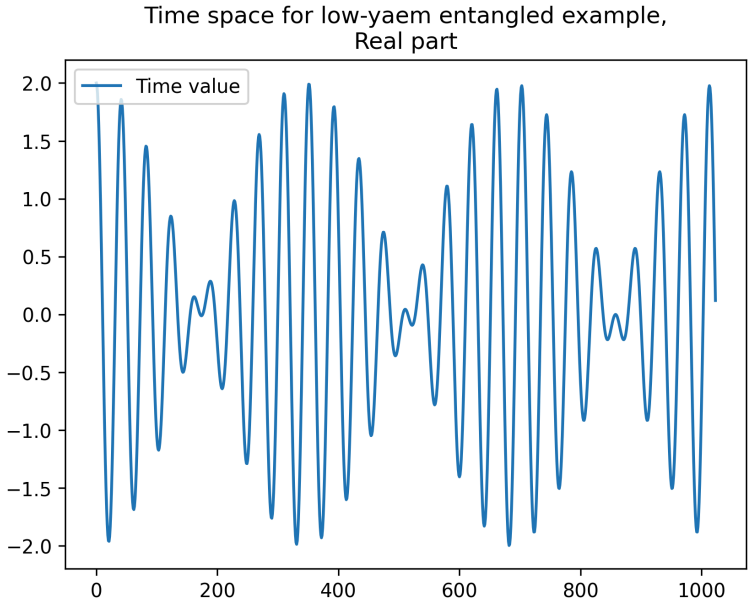
```
        res *= propbits[propbit_idx].dot(curr_fft_propbits[propbit_idx])
    return res

if __name__ == "__main__":
    test_period(
        39, 10, functools.partial(idx_fn_trivial_example, mod=39,
num_propbits=10)
    )
    test_period(
        39,
        10,
        functools.partial(
            idx_fn_low_yaem_entangled_example, mod1=39, mod2=51,
num_propbits=10
        ),
    )
    # Check all the periods this following one gives you
    test_period(
        15,
        10,
        functools.partial(
            idx_fn_marginally_less_trivial_example, mod=15,
num_propbits=10
        ),
    )
```

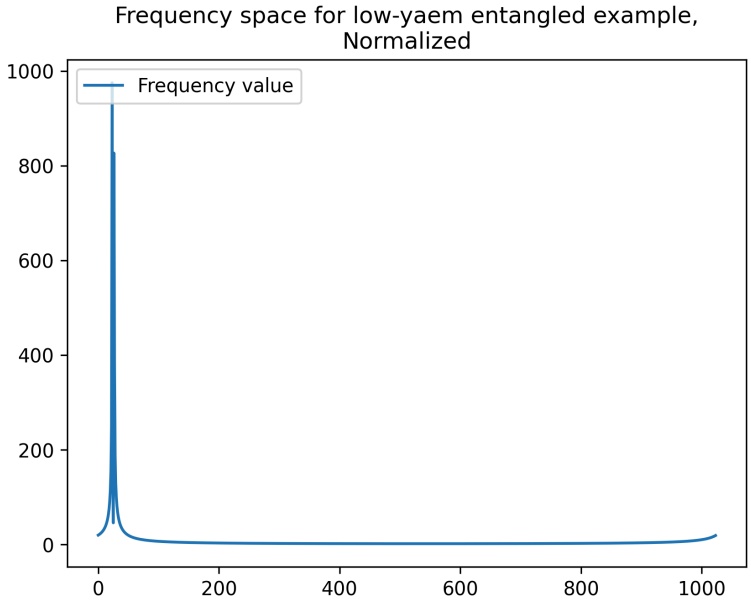



Frequency space for trivial example, normalized, x mod 39
Frequency value
800
600
400
200
0
0
200
400
600
800
1000


Time space for marginally less trivial example,
Real part, x mod 15
Time value
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
200
400
600
800
1000


Frequency space for marginally less trivial example,
Normalized, x mod 15
Frequency value
800
600
400
200
0
0
200
400
600
800
1000


Time space for low-yaem entangled example,
Real part
Time value
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
200
400
600
800
1000



To make the charts comparable, all spectra are for 10 propbits, but in reality one would wish for more margin of error on the amplitude vector sizes for the less trivial examples. This is a factor of 2 on number of propbitsets only.

Bennett Hyperoperation (Commutative Hyperoperation)

Consider the commutative hyperoperation of the 3rd degree, first considered by Bennett in 1915 [27]. Denote it as comhop-3 to ward off carpal tunnel. Despite its obscurity it is simple. Denote the comhop-3 operation by $\ominus$, because $\otimes$ is already taken for the Kronecker product.[10]

$$x \ominus y := \exp(\log(x) \times \log(y))$$

In order to get the modular exponents in a propbitset in an unentangled manner, we have to work in a different ring than the propbitsets we've been working with. The lesser operation of that ring is ordinary multiplication (as opposed to most other rings, where multiplication is the greater operation). The greater operation is $\ominus$.

[10] comhop-2 and comhop-1 are more quotidian: multiplication and addition, respectively. Therefore we will sometimes just call the $\ominus$ operator *comhop*, instead of comhop-3.

We claim that the two operations of multiplication and the comhop-3 on the reals constitute a ring straightforwardly. However, due to the behavior of the complex logarithm, in order to get multiplication and comhop-3 on the complex numbers to be a near-ring, the ring set elements must be circumscribed to the unit complex circle, with a formal algebraic difficulty because the identity element is not in the unit complex circle.

This is actually the *most* problematic concession we must make, because it prevents us from easily getting at the propbit-set fast fourier transform on the ring for modular exponentiation semantics, unlike how in the modular range ansatz we were able to get the fast fourier transform in a timely manner.[11]

[11] We also stated in the introduction that the set of feasible functions where we could find periods feasibly is different than the set of Gottesman-Knoll-compatible functions. This is from the change in the ring.

Multiplication and comhop-3 on the reals make a ring.
Recall that multiplication is associative and commutative. The lesser operation's identity is 1, not 0, but there is one. Lesser operation inverse of a is $\frac{1}{a}$, not $-a$, but it exists.

Comhop-3 is associative in the reals, because

$$\log\left(e^{\log a \times \log b}\right) = \log a \times \log b \tag{4}$$

in the reals, so

$$e^{\log\left(e^{\log a \times \log b}\right) \times \log c} = e^{\log a \times \log\left(e^{\log b \times \log c}\right)} \tag{5}$$

in the reals. The greater operation identity is e, not 1 as in addition-multiplication ring, but it is an identity (because $\log e = 1$, so we are multiplying in the exponent of the comhop-3 by 1).

In the reals, comhop-3 is distributed with respect to multiplication in a simple manner, because

$$e^{\log a \times (\log b + \log c)} = e^{\log a \times \log b} \times e^{\log a \times \log c} \tag{6}$$

. Hence a ring in the reals.

Complex picture
Recall that the real exponent is invertible via the real logarithm, whereas in the complex case the invertibility is modulo a winding, so the complex logarithm is not strictly a function.

We cannot be taking principal branches here, we need to really think about the winding, because it is that periodic semantics of the complex logarithm which we are dealing with.

Therefore, the picture with comhop-3 is fairly more miserable in the complex numbers, because the complex logarithm is a

more miserable de-facto-non-function than the real logarithm. Without taking care of all the winding, only a subset of the ring semantics only holds in the complex unit circle. Nonetheless, we can still use this to turn SO(2) from a group to a ring.[12]

[12] The 2-space special orthogonal group SO(2) is isomorphic to the circle group, where the operator on the group is adding angles together, so SO(2) can be imagined geometrically that way.

Imagining the algebra as a ring by inserting comhop-3 as the greater operation to this, then, corresponds to multiplying the angles (and therefore is subject to the many pitfalls of multiplying-angles semantics). The lesser operation is complex multiplication now, and still holds all the ring properties.

We need to constrain to the complex unit circle to still have the greater operation properties, as far as we can tell. If a, b are on the complex unit circle or are e, complex comhop-3 associates. This is because

$$\log\left(e^{\log a \times \log b}\right) = \log a \times \log b \tag{7}$$

still holds on the complex unit circle or with e, just as it did with the reals. This is not the case in the general complex plane.

We still have the same comhop-3 identity. Distributivity in the complex has the idiosyncratic property that it needs to be on an even number of propbits in a propbitset using the ring (that the thing only works for even numbers being distributed), because otherwise we shove ourselves out of the complex in doing the complex exponential. So strictly speaking we are a near-ring, but we can do distributivity for our purposes.

Given the ring, however, propbitsets with comhop-3 as the operator are straightforward.

```
import numpy as np

def comhop_op(*args):
    return np.exp(np.prod(np.array([np.log(arg) for arg in args])))

def comhop_idx_fn(propbits, idx):
    members = []
    idx_repr = list(map(int, np.binary_repr(idx, width=len(propbits))))
    for propbit_idx, propbit_val in enumerate(idx_repr):
        members.append(propbits[propbit_idx][propbit_val])
    return comhop_op(*members)
```

```
# Note that all these are on the
complex unit circle. You will have
to get closer if you wish for more
sigfigs
>> example_propbitset = [
  [0+1j, 0.5 + 0.866j],
  [0.4 + 0.916j, 0.4 + 0.916j]]
>> np.round(
  comhop_idx_fn(
    example_propbitset, 3), 8)
(0.29707693-0.00015448j)
```

Importantly, again, in the full complex space the algebra with multiplication and comhop-3 is not a ring, because comhop-3 does not associate in the general case. Try $(4, 3i)$, $(8, 3i)$ and $(5, 9i)$ as counterexample.

This seems a strange and trivial property to have to not hold for cryptographical apocalypse to not descend upon us like a thief in the night, and we have made some small attempts at getting the associativity to hold which end up expanding the yaem (and therefore the time for period-finding) exponentially, so getting the thing to work is not as trivial as all that. But this is indeed the property, as we shall see now.

One possible method of representing modular exponentiation using the complex number ansatz as a propbit goes like this. We have explored some trivial others but the unfortunate aspects of the representation remain the same, but we also have no proof that there is no better representation.

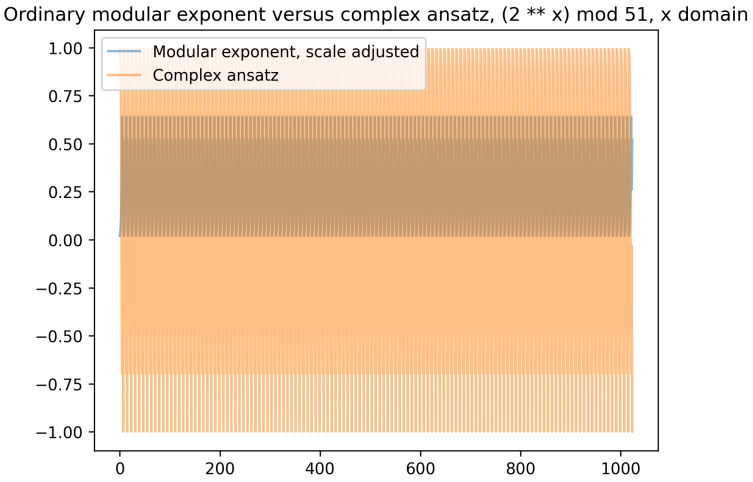
```
def complex_modexp_repr(num_pbs, mod):
    # Replace base if different base
    # Modulo mainly to do overflows
    sequence = [pow(2, 2 ** n, mod) for n in range(num_pbs - 1, -1, -1)]

    fst = [np.exp(2j * np.pi / mod), np.exp(2j * np.pi * sequence[0] /
mod)]
    rest = [[np.e, np.exp(sequence_member)] for sequence_member in
sequence[1:]]
    res = [fst] + rest
    return np.array(res)
```

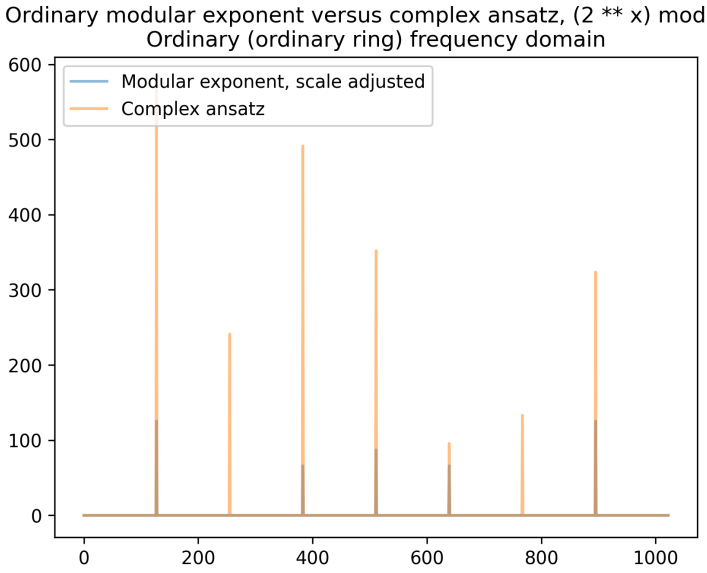
This is derived by thinking about the equivalent of the geometric series in doubly-exponential space by comhop-3-factorization, as opposed to ordinary multiplicative factorization in a singly-exponential space.

Note that the members of the non-initial propbits are not on the complex unit circle. But this state can be constructed with one propbitset, so it seems unentangled, for whatever definition of entanglement induced by the ring.

The placement of the non-initial propbits in real space is to avoid the complex winding semantics of repeated flitting in and out of complex logarithms.

```
>> mod = 11
>> np.array([[np.exp(2j * np.pi /
mod), np.exp(2j * np.pi * (2 ** 8) /
mod)],
        [np.e, np.exp(2 ** 4)],
        [np.e, np.exp(2 ** 2)],
        [np.e, np.exp(2 ** 1)]])
array([[ 8.4125e-01+0.5406j,
-1.4231e-01+0.9898j],
       [ 2.7183,  8.8861e+06],
       [ 2.7183,  5.4598e+01],
       [ 2.7183,  7.3891e+00]])

>> complex_modexp_repr(4, 11)
array([[ 8.4125e-01+0.5406j
-1.4231e-01+0.9898j]
 [ 2.7183, 8.8861e+06]
 [ 2.7183, 5.4598e+01]
 [ 2.7183, 7.3891e+00]])
# Growth being too big can be fixed
by a modulo to `sequence`
members
```





Note that the notion of linearity is defined with respect to a ring. Ordinary linearity of a function f is such that:

$$f(x+y) = f(x) + f(y) \tag{8}$$

.

and

$$f(ax) = a(f) \forall a \tag{9}$$

,

but this is with respect to a lesser (+) and greater ($\times$) operation, with variables in some vector space. With our comhop-3

ring, we could define a comhop-3-based linearity of a function f, where a comhop-3-linear function f would satisfy:

$$f(x * y) = f(x) * f(y) \tag{10}$$

and

$$f(a \ominus x) = a \ominus f(x) \tag{11}$$

. Not at all synonymous with ordinary linearity.

The notion of an inner product is with respect to a vector space. Vector spaces are modules with respect to a field. We have a different ring, and therefore a whole different inner product, defined by simply swapping out the lesser and greater ring operations in the inner product. We previously ignored the division operation, and in fact we seem to be working in a situation where we can actually just treat everything like a module. These may be famous last words.

There is one implementation detail which is pretty important, noted in the code section below.

```
import numpy as np

def comhop_inner_product(fst_pbset, snd_pbset):
    assert len(fst_pbset) == len(snd_pbset)
    res_inner = 1. + 0j
    # note that all the logarithm multiplication has to be done batched,
and one single complex exponent at the end, even in the complex unit
circle, because of the poor behavior with respect to associativity
    for pb_idx in range(len(fst_pbset)):
        res_inner *= (
          (np.log(fst_pbset[pb_idx][0]) * np.log(snd_pbset[pb_idx][0])) +\
          (np.log(fst_pbset[pb_idx][1]) * np.log(snd_pbset[pb_idx][1]))
        )
    return np.exp(res_inner)

def part_expand_ch(propbits):
    """Defined for testing"""
    def idx_fn(idx):
        members = []
        idx_repr = list(map(int, np.binary_repr(idx,
width=len(propbits))))
        for propbit_idx, propbit_val in enumerate(idx_repr):
            members.append(propbits[propbit_idx][propbit_val])
        return np.prod(np.log(members))
    return idx_fn
```

```
def full_fn_expand(idx_fn,
num_propbits):
    return np.array(
      [idx_fn(idx)
      for idx in
      range(int(2 ** num_propbits))]
    )

>> fst_pbset = [[np.exp(7.2j),
np.exp(-10.2j)],
          [np.exp(0.3j), np.exp(5.5j)],
          [np.exp(-10j), np.exp(1.5j)],
          [np.exp(-6.3j), np.exp(-6.2j)]]
>> snd_pbset = [[np.exp(0.1j),
np.exp(2.5j)],
          [np.exp(17.3j), np.exp(6.1j)],
          [np.exp(-1j), np.exp(2.8j)],
          [np.exp(3.3j), np.exp(8.1j)]]
>> fst_expanded =
full_fn_expand(part_expand_ch(fst_pbset),
len(fst_pbset))
>> snd_expanded =
full_fn_expand(part_expand_ch(snd_pbset),
len(snd_pbset))

>> full_inner_product_res =
np.prod(np.exp(fst_res * snd_res))
(0.5299698902585424+1.4824643551249408e-16j)

>> our_inner_product_res =
comhop_inner_product(fst_pbset,
snd_pbset)
(0.5299698902585424+1.4824643551249408e-16j)
```

With the notion of an inner product, we also have a comhop-linear notion of a Hadamard operator, cognate to the ordinary linear Hadamard operator.

```
import numpy as np
import scipy.linalg as sci_lin

def comhop_op(*args):
    return np.exp(np.prod(np.array([np.log(arg) for arg in args])))

def comhop_propbits(propbits):
    def idx_fn(idx):
        members = []
        idx_repr = list(map(int, np.binary_repr(idx,
width=len(propbits))))
        for propbit_idx, propbit_val in enumerate(idx_repr):
            members.append(propbits[propbit_idx][propbit_val])
        return comhop_op(*members)
    return idx_fn

def comhop_had(propbits):
    had_propbits = []
    for propbit in propbits:
        # Recall that the lesser operation in this ring is multiplication
        had_propbits.append([propbit[0] * propbit[1], propbit[0] /
propbit[1]])
    return comhop_propbits(had_propbits)

# Dealing with the corrections
def make_correction(propbit_idx, correction_location, log_had_res):
    # before
    correction = []
    for before in log_had_res[:propbit_idx]:
        correction.append(before)
    # member
    if correction_location[0] == -1:
        correction.append(np.array([-2j * np.pi, 0]))
    elif correction_location[0] == 1:
        correction.append(np.array([2j * np.pi, 0]))
    elif correction_location[1] == -1:
        correction.append(np.array([0, -2j * np.pi]))
    elif correction_location[1] == 1:
        correction.append(np.array([0, 2j * np.pi]))
```

```
def full_fn_expand(idx_fn,
num_propbits):
    return np.array(
      [idx_fn(idx)
      for idx in
      range(int(2 ** num_propbits))]
    )

>> pbset = [[np.exp(3.3j),
np.exp(-15.5j)],
          [np.exp(-6.3j), np.exp(-6.2j)]]
>> pbset_expanded =
full_fn_expand(comhop_propbits(pbset),
len(pbset))
>> pbset_expanded_had =
test_comhop_had_apply(pbset_expanded)
[1.48097792+3.16086871e-16j
0.55339598+1.76509870e-16j
 1.00329447-1.66227529e-18j
0.99505667+7.82219177e-18j]

>> comhop_had_idx_fn =
comhop_had(pbset)
>> comhop_had_expanded =
full_fn_expand(comhop_had_idx_fn,
len(pbset))
[0.97597702-4.55770240e-18j
1.03731647-7.38045905e-17j
 1.00329447-5.96456808e-18j
0.99505667+1.94362154e-17j]
# Note the discrepancy, which we
have to fix with the corrected
version that deals with winding,
which is more yaems, but not
exponentially more, as far as we
can tell

>> corrected_idx_fn =
corrected_comhop_had(pbset)
>> corrected_had_expanded =
full_fn_expand(corrected_idx_fn,
len(pbset))
[1.48097792+3.30553748e-16j
0.55339598+1.47046103e-16j
 1.00329447-5.96456808e-18j
0.99505667+1.94362154e-17j]
```

```
    else:
        raise Exception("invalid correction location")
    # after
    for remaining in log_had_res[propbit_idx + 1:]:
        correction.append(remaining)
    return correction

def corrected_comhop_had(cexp_propbits):
    """Not full general case, but can be corrected to full general case, we
think"""
    had_propbits = [[pb[0] * pb[1], pb[0] / pb[1]] for pb in
cexp_propbits]
    log_had_propbits = [[np.log(pb[0]) + np.log(pb[1]), np.log(pb[0]) -
np.log(pb[1])] for pb in cexp_propbits]
    print("log had propbits: ", log_had_propbits)
    def idx_fn(idx):
        corrections = []
        members = []
        idx_repr = list(map(int, np.binary_repr(idx,
width=len(had_propbits))))
        for propbit_idx, propbit_val in enumerate(idx_repr):
            members.append(had_propbits[propbit_idx][propbit_val])
            if log_had_propbits[propbit_idx][0].imag < -np.pi:
                corrections.append(make_correction(propbit_idx, [-1, 0],
log_had_propbits))
            elif log_had_propbits[propbit_idx][0].imag > np.pi:
                corrections.append(make_correction(propbit_idx, [1, 0],
log_had_propbits))
            elif log_had_propbits[propbit_idx][1].imag < -np.pi:
                corrections.append(make_correction(propbit_idx, [0, -1],
log_had_propbits))
            elif log_had_propbits[propbit_idx][1].imag > np.pi:
                corrections.append(make_correction(propbit_idx, [0, 1],
log_had_propbits))
        members_res = comhop.comhop_op(*members)
        corrected_res = members_res
        # Execute corrections
        for correction in corrections:
            correction_members = []
            for propbit_idx, propbit_val in enumerate(idx_repr):
                correction_members.append(correction[propbit_idx]
[propbit_val])
            # the 2pi winding reset messes with us otherwise
            corrected_res *= np.exp(np.prod(correction_members))
```

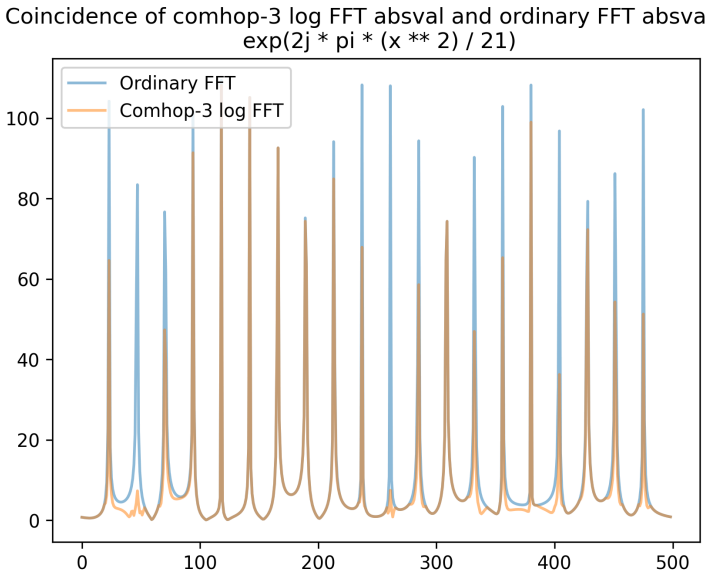
```
        return corrected_res
    return idx_fn

def test_comhop_had_apply(arr):
    """For testing purposes"""
    had_arr = sci_lin.hadamard(arr.shape[0])
    return np.prod(np.exp(np.log(arr) * had_arr), axis=1)
```

Fourier transform is somewhat fiddlier algebraically, because the roots of unity for the Fourier transform we are thinking of are not the roots of unity with respect to the greater-operation identity, which is e, but they are roots of unity with respect to 1. [13]

[13] You do need to do the corrections and stuff for the incidentally created windings in Fourier transform, just as in Hadamard transform. Outside of the SO(2) ring things get complicated. Omitted out of laziness, pique and incredible-delays-in-getting-the-Hadamard-thing-in.

```
import numpy as np
import numpy.fft as np_fft

def unity_root(m):
    """ 2 ** mth root of unity, so we can just put in num propbits as m
"""
    return np.exp((2j * np.pi) / (2 ** m))

def comhop_fft_pbs(num_propbits, idx):
    fft_pbs = [np.array([0, 0]) for _ in range(num_propbits)]
    for propbit_idx in range(num_propbits):
        exp = -(2 ** (num_propbits - propbit_idx - 1)) * idx
        fft_pbs[propbit_idx] = np.array([np.e + 0j,
np.exp(unity_root(num_propbits) ** exp)])
    return fft_pbs

def test_comhop_fft_apply(arr):
    """For testing"""
    fft_mat = np_fft.fft(np.eye(arr.shape[0]))
    return np.prod(np.exp(np.log(arr) * fft_mat), axis=1)

def fft_inner(pbs):
    def idx_fn(idx):
        fft_pbs = comhop_fft_pbs(len(pbs), idx)
        res_inner = 1. + 0j
        for pb_idx in range(len(pbs)):
            res_inner *= ((np.log(pbs[pb_idx][0]) * np.log(fft_pbs[pb_idx]
[0])) + (np.log(pbs[pb_idx][1]) * np.log(fft_pbs[pb_idx][1])))
        return np.exp(res_inner)
    return idx_fn
```

Why we would want such a definition is because the peaks of this comhop Fourier transform seem to correspond to the peaks of the ordinary Fourier transform of the amplitude vector. This is regrettably not proof, but empirics. Lots of room for fiddling around here, and we have fiddled around with much of it, but we make no claims as to completeness in our fiddling. Fiddle away if you wish.



Complex ansatz on modular exponent, problems

So, using the comhops, we are able to get the semigroup but not the ring semantics to work on the propbitsets. What is meant by this is that, in order to get a propbitset representing our complex exponent ansatz on modular exponent, we have some propbitsets that look like so:

$$\left[\left[e^{\frac{2i\pi}{n}}, e^{\frac{2i\pi(2^m)}{n}}\right], \left[e, e^{2^{m-1}}\right], \left[e, e^{2^{m-2}}\right], \ldots \left[e, e^{2^1}\right]\right]$$

where m is number of propbits and n is the modulo.

But, importantly, they have now gotten out of the complex unit circle where we had associativity, because of those real members in the non-first propbits.

This is a problem, because what we have of the working Fourier transform on the comhop ring does depend upon that associativity. Therefore, the pseudo-quantum Fourier transform on the comhop ring works on a propbitset that stays in the complex unit circle, but does not work on the complex representation corresponding to the modular exponent.

So we fundamentally have a fairly miserable analytic continuation in front of us, miserable inasmuch as we must try to represent this polynomially in order to have the overall algorithm be relevant. We are not really cognizant of any serious compu-

tational treatments of the complex algebra and of the analytic continuation in general.

Complex ansatz on modular square

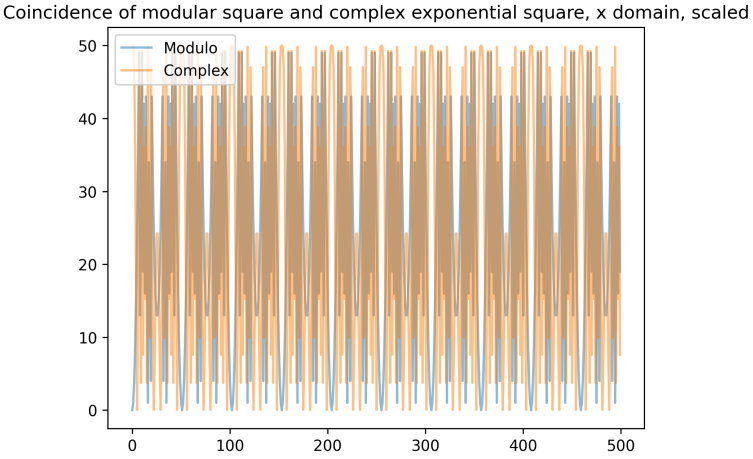
An aside. We could do the complex ansatz on modular squaring. Note that we could do congruence of squares still with this, and empirically on small N it seems to us that there's an extensive number of nontrivial congruences of squares with modular squaring.

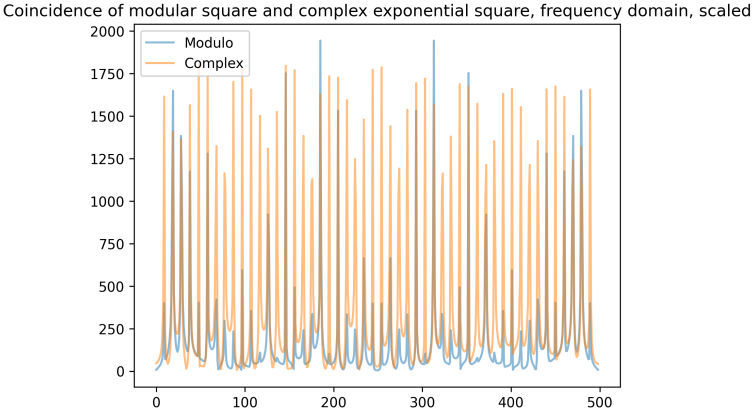
We give the complex ansatz on modular squaring here below.

```
import numpy as np
import copy

def comhop_op(*args):
    return np.exp(np.prod(np.array([np.log(arg) for arg in args])))

def comhop_propbits(propbits):
    def idx_fn(idx):
        members = []
        idx_repr = list(map(int, np.binary_repr(idx,
width=len(propbits))))
        for propbit_idx, propbit_val in enumerate(idx_repr):
            members.append(propbits[propbit_idx][propbit_val])
        return comhop_op(*members)
    return idx_fn

def comhop_modular_squaring_propbitsets(num_pbs, mod):
    total_pbs = []
    for curr_pbset_idx in range(num_pbs):
        curr_pbset = [[np.e, np.e] for _ in range(num_pbs)]
        curr_pbset[curr_pbset_idx] = [1, np.e]
        for pb_idx in range(num_pbs):
            curr_pbset_2 = copy.deepcopy(curr_pbset)
            mult = 2 ** (num_pbs * 2 - curr_pbset_idx - pb_idx - 2)
            curr_pbset_2[pb_idx] = [1, np.exp(2j * np.pi * mult / mod)]
            total_pbs.append(curr_pbset_2)
    return total_pbs
```

Derive by taking the propbitset representation of ordinary squares and translating to the multiplication-and-commutative-hyperoperation ring.

```
def full_fn_expand(idx_fn,
num_propbits):
    return np.array(
      [idx_fn(idx)
      for idx in
      range(int(2 ** num_propbits))]
    )

>> mod = 19
>> correct_res = np.exp(2j * np.pi *
(np.linspace(0, 15, 16) ** 2) / mod)
array([ 1.        +0.j        ,
0.94581724+0.32469947j,
        0.24548549+0.96940027j,
-0.9863613 +0.16459459j,
        0.54694816-0.83716648j,
-0.40169542+0.91577333j,
        0.78914051-0.61421271j,
-0.87947375-0.47594739j,
        -0.67728157+0.73572391j,
-0.08257935+0.99658449j,
        -0.08257935+0.99658449j,
-0.67728157+0.73572391j,
        -0.87947375-0.47594739j,
0.78914051-0.61421271j,
        -0.40169542+0.91577333j,
0.54694816-0.83716648j])
>> pbsets =
comhop_modular_squaring_propbitsets(4,
mod)
>> expansions =
[full_fn_expand(comhop_propbits(pbset),
4) for pbset in pbsets]
>> np.prod(expansions, axis=0)
array([ 1.        +0.j        ,
0.94581724+0.32469947j,
        0.24548549+0.96940027j,
-0.9863613 +0.16459459j,
        0.54694816-0.83716648j,
-0.40169542+0.91577333j,
        0.78914051-0.61421271j,
-0.87947375-0.47594739j,
        -0.67728157+0.73572391j,
-0.08257935+0.99658449j,
        -0.08257935+0.99658449j,
-0.67728157+0.73572391j,
        -0.87947375-0.47594739j,
0.78914051-0.61421271j,
        -0.40169542+0.91577333j,
0.54694816-0.83716648j])
```





We have sort of bounced off of doing pseudo-quantum Fourier transform with respect to these, and especially combining the propbitsets which make up this function together without an exponential explosion of yaems. However we have only spent a few weeks on this specific one so have at it if you are inclined.

The immediate obvious problem is that the lesser operation (multiplication) is more difficult to combine members of with respect to comhop-linearity than addition with respect to ordinary linearity, in the complex numbers.

Reasons to believe fast modular exponent Fourier transform with the ansatz is possible

- We have a solid implementation of the trivial modular range with Fourier transform, so the ansatz does work for trivial cases.
- We have the propbitset representation of the complex ansatz on the modular exponent working fine in and of itself: the Fourier transform is what's missing.
- Comhop-3 Hadamard transforms work fine, even on the complex ansatz on the modular exponent, because they only use the lesser operation in the ring so they do not hit upon the problem with associativity of comhop-3 in complex space. Even though we cannot immediately find a way to do period finding over the field we want with the Hadamard transform, it becomes difficult to think that other integral transforms can be so much more radically difficult than the Hadamard transform.
- Everything works fine on SO(2) (or, rather, the ring which we generate by slapping comhop-3 onto SO(2) as the greater operation), so there is an algebraic structure where this all works, and the complex ansatz for the modular exponent busting out of SO(2) is the problem.
- We still have a lot of formal room possible in retreating again to bicomplex numbers, Montgomery reductions on propbitsets and things of that nature.
- However, we cannot colorably give a quick resolution of the question because we have gone and gotten a vaguely normal job touching computers normal-like again.

Misc

There has been a movement towards postquantum cryptography in anticipation of someone making quantum computers. We believe that this document also gives some evidence that there might be a possibility of attacking hidden abelian subgroup cryptography even without actual quantum computers.

This is important because any rando lying around eating burritos with a laptop, like the one writing this document, can do research on this *specific* subject in the classical domain.

Attempts at quantum computation by actual quantum computers require hundreds of millions of dollars, dozens to hundreds of staff, and entire institutions dealing with that sort of thing. There was, therefore, never any danger of fully-clandestine non-state quantum computing, but clandestine classical computing by complete rando private individuals is routine. Advanced-persistent-threat-level actors are capable of trying both sorts of things clandestinely but what can you do?[14]

If we go on to find a suitable attack on hidden abelian subgroups, we claim that we will announce it in a timely and public manner without exploitation. No such guarantee is given to any of us by other parties. Any such break presents a veritable ring of Gyges to the one who figures out the break. Make decisions with regards to this claim as you will.

Ultimately, our conjecture is that trapdoor functions do not actually exist and that asymmetric cryptography will be shown to be ultimately impossible, whether based upon abelian or dihedral subgroups or whatever set of subgroups one would like.[15] This is pretty important if you recall the gigantic data stores in Utah that the NSA has of substantively all encrypted data anywhere and the probable adversarial backups the PLA has almost surely made, probably in Zhongguancun somewhere, because the NSA is pretty great at attacking and pretty worthless at defending. [16]

We have no colorable way to prove to you or convince you that we did this substantively for the humor value, but that was why we did it. We have no idea if the grantor also was motivated by this.[17]

We regret to note that if you have already read up to here, there is an implied permission you have granted for us to do some violence to your eyeballs. We thought up of and wrote this other dealio, so the implicature goes that we can subject your eyeballs to it. Have yet another relativization-avoiding diagonalization (https://github.com/howonlee/yarad).

[14] J. Mickens[28] suggests, we are sure in absolute grim seriousness and with no tongue in cheek whatsoever, magic amulets or moving to a submarine. You do you.

[15] We predict that peeps will actually prove this with concrete implementations in a timely manner for what it is - between two days and two centuries.

[16] And vice versa: the PRC official PKI requires key escrow with the government, and therefore also probable escrow at the NSA because they really are great at attack.

[17] Our previous highest-effort attempt at a joke was via making 4-figures in (legal) political contributions over a few weeks and then prank replying to the senatorial candidate when they called asking for more money, so this breaks our personal record by an order of magnitude in money, effort and time. Perhaps better in general worthwhile character of produced artifact, perhaps not.

We have many, many more failed attempts, which you can contact us directly if you're interested in listening to us expound upon them at serious length. A summary would mention attempts at doing the number system shift again with quaternions and bicomplex numbers (tessarines) [29], some attempted shenanigans with Montgomery's [30], more complex-analytic attempts with some actual analysis, attempts to prove properties about winding number structures, trying to do the whole thing via just Hadamard transform instead of full Fourier transform in a Simon's-like way, and just inverting the thing without the Fourier semantics at all, what with there obviously being no no-cloning theorem [31] for this thing, among many others.

Generally, we believe the best path forward is bicomplex numbers or something similar to them, perhaps by a Cayley-Dickson construction or not[18]. The special attention to the bicomplex numbers is because of the symmetry in the construction with respect to the ordinary complex numbers, which quaternions abandon in order to be able to model 3-space, which we do not care about. [19]

However, we are not well-resourced enough to actually spend enough time exploring composition algebras at serious depth and the original grant was basically on a drunken lark. As mentioned before, our political credibility for working in the field is poor and we intend to keep it that way, and therefore formal collaboration and regular grant-getting is probably not feasible for us.

We have gone and gotten another software-touching job and plan to go back to doing this only on the weekends, so if you wish to try to scoop us on getting a cryptographical apocalypse going or prove some kind of class separation showing that this approach is impossible for the modular exponent case, you are welcome to go ahead and do that.

Thanks to the irregular anonymous grantor. [20] With regards to the circumstances of the grant we have promised discretion.

Thanks to JB, PAB and AJB for eating my baking and candy.

[18] Some kind of obscure composition algebra, anyways.

[19] An extreme non-sequitur: Onsager's original proof of the 2D Ising model phase transition [32] was based upon sums of iterated direct products on quaternion algebras, and direct products are easily seen to be very related to Kronecker products. Obviously peeps get taught different proofs of the 2D Ising model transition nowadays but we believe that this is of interest.

[20] Although we are pretty sure that genuine apocalypse cultists do not actually call themselves apocalypse cultists. Sorry for wheedling so long to be allowed to reveal the existence of the grant, although that was mainly to try to get credibility for the next job.

Non-Sequitur Wittgenstein Quotation

1. The world is all that is the case.

1.1 The world is the totality of facts, not of things.

1.11 The world is determined by the facts, and by their being all the facts.

1.12 For the totality of facts determines what is the case, and also whatever is not the case.

1.13 The facts in logical space are the world.

1.2 The world divides into facts.

1.21 Each item can be the case or not the case while everything else remains the same.

— L. Wittgenstein [33]

References

[1] L. B. Torvalds, [Online]. Available: https://lwn.net/Articles/193245/

[2] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, doi: 10.1137/s0097539795293172.

[3] C. Lomont, "The Hidden Subgroup Problem - Review and Open Problems," 2004, [Online]. Available: https://arxiv.org/abs/quant-ph/0411037

[4] A. Barenco *et al.*, "Elementary gates for quantum computation," *Phys. Rev. A*, vol. 52, no. 5, pp. 3457–3467, Nov. 1995, doi: 10.1103/PhysRevA.52.3457.

[5] D. Gottesman, "The Heisenberg Representation of Quantum Computers," 1998, [Online]. Available: https://arxiv.org/abs/quant-ph/9807006

[6] D. R. Simon, "On the Power of Quantum Computation," *SIAM Journal on Computing*, vol. 26, no. 5, pp.

1474–1483, 1997, doi: 10.1137/S0097539796298637.

[7] T. mpmath development team, “mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.3.0).” 2023.

[8] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[9] X. Xu, S. Benjamin, J. Sun, X. Yuan, and P. Zhang, “A Herculean task: Classical simulation of quantum computers.” [Online]. Available: https://arxiv.org/abs/2302.08880

[10] K. Svozil, *Quantum logic*. Springer Science & Business Media, 1998.

[11] *Advances in Linear Logic*. in London Mathematical Society Lecture Note Series. Cambridge University Press, 1995.

[12] R. Orús, “Tensor networks for complex quantum systems,” *Nature Reviews Physics*, vol. 1, no. 9, pp. 538–550, 2019.

[13] K. Schacke, “On the kronecker product,” *Master's thesis, University of Waterloo*, 2004.

[14] A. Satriawan, I. Syafalni, R. Mareta, I. Anshori, W. Shalannanda, and A. Barra, “Conceptual review on number theoretic transform and comprehensive review on its implementations,” *IEEE Access*, 2023.

[15] M. B. Plenio and S. Virmani, “An introduction to entanglement measures.” [Online]. Available: https://arxiv.org/abs/quant-ph/0504163

[16] Z.-C. Yang, A. Hamma, S. M. Giampaolo, E. R. Mucciolo, and C. Chamon, “Entanglement complexity in quantum many-body dynamics, thermalization, and localization,” *Phys. Rev. B*, vol. 96, no. 2, p. 20408, Jul. 2017, doi: 10.1103/PhysRevB.96.020408.

[17] P. Duhamel and M. Vetterli, “Fast Fourier transforms: a tutorial review and a state of the art,” *Signal processing*, vol. 19, no. 4, pp. 259–299, 1990.

[18] D. Coppersmith, “An approximate Fourier transform useful in quantum factoring.” [Online]. Available: https://arxiv.org/abs/quant-ph/0201067

[19] D. E. Browne, “Efficient classical simulation of the quantum Fourier transform,” *New Journal of Physics*, vol. 9, no. 5, p. 146, May 2007, doi: 10.1088/1367-2630/9/5/146.

[20] S. S. Agaian, H. Sarukhanyan, K. Egiazarian, and J. Astola, “Hadamard transforms,” 2011.

[21] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, “Simple and practical algorithm for sparse Fourier transform,” in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, 2012, pp. 1183–1194.

[22] S. Aaronson, “Is quantum mechanics an island in theoryspace?,” *arXiv preprint quant-ph/0401062*, 2004.

[23] E. Demaine, “Lecture 2: Introduction to Algorithms.” [Online]. Available: https://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf

[24] P. Virtanen *et al.*, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.

[25] L. Ruiz-Perez and J. C. Garcia-Escartin, “Quantum arithmetic with the quantum Fourier transform,” *Quantum Information Processing*, vol. 16, pp. 1–14, 2017.

[26] M. Guštin, *TheMatjaz/contfrac*. 2019. [Online]. Available: https://github.com/TheMatjaz/contfrac

[27] A. A. Bennett, “Note on an Operation of the Thrid Grade,” *Annals of Mathematics*, vol. 17, no. 2, pp. 74–75, 1915.

[28] J. Mickens, “This World of Ours.” [Online]. Available: https://www.usenix.org/system/files/1401_08-12_mickens.pdf

[29] M. Luna-Elizarraras, M. Shapiro, D. C. Struppa, and A. Vajiac, “Bicomplex numbers and their elementary functions,” *Cubo (Temuco)*, vol. 14, no. 2, pp. 61–80, 2012.

[30] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[31] W. K. Wootters and W. H. Zurek, “A single quantum cannot be cloned,” *Nature*, vol. 299, no. 5886, pp. 802–803, 1982.

[32] L. Onsager, “Crystal statistics. I. A two-dimensional model with an order-disorder transition,” *Physical Review*, vol. 65, no. 3–4, p. 117, 1944.

[33] L. Wittgenstein, “Tractatus Logico-Philosophicus.” JSTOR, 1923.