With Fast Fourier transforms it's possible to convert a set of evaluations of a polynomial over a prime field at a set of specific points, $P(1)$

, $P(r)$

, $P(r^2)$

… $P(r^{{2^k}-1})$

(where $r^{2^k} = 1$

) into the polynomial's coefficients in $n * \log(n)$

time. This is used extensively and is key to the efficiency of [STARKs](#) and most other general-purpose ZKP schemes. Polynomial interpolation (and its inverse, multi-point evaluation) is also

used extensively in erasure coding, which is useful for [data recovery](#) and in blockchains for [data availability checking](#).

Unfortunately, the FFT interpolation algorithm works only if you have all

evaluations at $P(r^i)$

for $0 \le i \le 2^k - 1$

. However, it turns out that you can make a somewhat more complex algorithm to also achieve polylog complexity for interpolating a polynomial (ie. the operation needed to recover the original data from an erasure code) in those cases where some of these evaluations are missing. Here's how you do it.

Erasure code recovery in $O(n*\log^2(n))$

time

Let $d[0] ... d[2^k - 1]$

be your data, substituting all unknown points with 0. Let $Z_{r, S}$

be the minimal polynomial that evaluates to 0 at all points r^k

for $k \in S$

. Let $E(x)$

(think E = error) be the polynomial that evaluates to your data with erasures (ie. $E(r^i) = d[i]$

), and let $P(x)$

be the polynomial that evaluates to original data. Let I

be the set of indices representing the missing points.

First of all, note that $D * Z_{r,I} = E * Z_{r,I}$

. This is because D

and E

agree on all points outside of I

, and $Z_{r,I}$

forces the evaluation on points inside

I

to zero. Hence, by computing $d[i] * Z_{r,I}(r^i) = (E * Z_{r,I})(r^i)$

, and interpolating $E * Z_{r,I}$

, we get $D * Z_{r,I}$

.

Now, how do we extract D

? Just evaluating pointwise and dividing won't work, as we already know that at least for some points (in fact, the points we're trying to recover!) $(D * Z_{r,I})(x) = Z_{r,I}(x) = 0$

, and we won't know what to put in place of $0 / 0$

. Instead, we do a trick: we generate a random k

, and compute $Q1(x) = (D * Z_{r,I})(k * x)$

(this can be done by multiplying the ith coefficient by k^{-i}

). We also compute $Q2(x) = Z_I(k * x)$

in the same way. Now, we compute $Q3 = Q1 / Q2$

(note: $Q3(x) = D(k * x)$

), and then from there we multiply the ith coefficient by k^i

to get back $D(x)$

, and evaluate $D(x)$

to get the missing points.

Altogether, outside of the calculation of $Z_{r,I}$

this requires six FFTs: one to calculate the evaluations of $Z_{r,I}$

, one to interpolate $E * Z_{r,I}$

, two to evaluate $Q1$

and $Q2$

, one to interpolate $Q3$

, and one to evaluate D

. The bulk of the complexity, unfortunately, is in a seemingly comparatively easy task: calculating the $Z_{r,I}$

polynomial.

Calculating Z in O(n*log^2(n))

time

The one remaining hard part is: how do we generate $Z_{r,S}$

in n*polylog(n) time? Here, we use a recursive algorithm modeled on the FFT itself. For a sufficiently small S

, we can compute $(x - r^{s_0}) * (x - r^{s_1}) ...$

explicitly. For anything larger, we do the following. Split up S

into two sets:

$S_{even} = {\frac{x}{2}\ for\ x \in S\ if\ S\ mod\ 2 = 0}$

$S_{odd} = {\frac{x-1}{2}\ for\ x \in S\ if\ S\ mod\ 2 = 1}$

Now, recursively compute $L = Z_{r^2, S_{even}}$

and $R = Z_{r^2, S_{odd}}$

. Note that L

evaluates to zero at all points $(r^2)^{\frac{s}{2}} = r^s$

for any even $s \in S$

, and R

evaluates to zero at all points $(r^2)^{\frac{s-1}{2}} = r^{s-1}$

for any odd s \in S

. We compute R'(x) = R(x * r)

using the method we already described above. We then use FFT multiplication (use two FFTs to evaluate both at 1, r, r^2 ... r^{-1}

, multiply the evaluations at each point, and interpolate back the result) to compute L * R'

, which evaluates to zero at all the desired points.

In one special case (where S

is the entire

set of possible indices), FFT multiplication fails and returns zero; we watch for this special case and in that case return the known correct polynomial, P(x) = x^{2^k} - 1

.

Here's the code that implements this (tests here):

github.com

ethereum/research/blob/master/mimc_stark/recovery.py

from fft import fft, mul_polys

Calculates modular inverses [1/values[0], 1/values[1] ...]

def multi_inv(values, modulus): partials = [1] for i in range(len(values)): partials.append(partials[-1] * values[i] % modulus) inv = pow(partials[-1], modulus - 2, modulus) outputs = [0] * len(values) for i in range(len(values), 0, -1): outputs[i-1] = partials[i-1] * inv % modulus inv = inv * values[i-1] % modulus return outputs

Generates q(x) = poly(k * x)

def p_of_kx(poly, modulus, k): o = [] power_of_k = 1 for x in poly: o.append(x * power_of_k % modulus)

This file has been truncated. show original

Questions:

- Can this be easily translated into binary fields?
- The above algorithm calculates Z_{r,S}

in time O(n * log^2(n))

. Is there a O(n * log(n))

time way to do it?

- If not, are there ways to achieve constant factor reductions by cutting down the number of FFTs per recursive step from 3 to 2 or even 1?
- Does this actually work correctly in 100% of cases?
- It's very possible that all of this was already invented by some guy in the 1980s, and more efficiently. Was it?