Pseudocode and Proofs: Exponential Mechanism with Base-2 Differential Privacy

Christina Ilvento

Contents

1 Overview

This document describes the proofs for the exact implementation of the exponential mechanism. An exact implementation of the exponential mechanism has significant benefits from the perspective of reducing floating-point vulnerabilities. The exact implementation is based on base-2 differential privacy (see Section 2)

There are four main pieces of the implementation:

1. Normalized sampling without division
2. Base-2 privacy parameters
3. Exact arithmetic helper methods and monitoring
4. Mechanism logic

1.1 Review History

Reviewer: Salil Vadhan, in progress (8/31/2020)

1.2 Additional documentation.

Full paper. The full paper describing this project can be found at https://arxiv.org/abs/1912.04222. This paper has been peer reviewed. **Aside:** Will add a link to peer reviewed version when available.

github. A version of this code is available on github: https://github.com/cilvento/b2dp. There may be minor differences in the public github version (vs the pull request), as the public github version includes development for additional projects.

1.3 General Notes and Caveats

Types. In general, we will refer to Rust types, including the types provided by the `Rug` crate e.g. `u32`, `Float`.

Differences from published versions. The pseudocode and proofs presented in this document are not intended to be identical to the published versions. We present pseudocode in a form that more closely matches the code to make verification easier.

[CRITICAL] Randomness Source. As always, the source of randomness used is critical to any guarantee of privacy. We provide a wrapper for OpenSSL generated randomness for convenience, but it is the caller's responsibility to ensure that the rng they provide is sufficiently secure.

[CRITICAL] Dependencies. The `Rug` and `gmp-mpfr-sys` crates are assumed to work as described. If there are errors in these crates, the guarantees of this code may be broken. In particular, if the arithmetic error flags are set improperly, there would be serious failures. A canary test exists for this functionality, but updates to dependencies should be checked carefully. (See Section 9.)

Thread safety/concurrency. This code has not been tested specifically for thread safety or concurrency issues.

2 Preliminaries

For reference, we include the specification of base-2 differential privacy as well as the desired output distribution of the base-2 exponential mechanism.

Definition 2.1 ($|_2$ Differential Privacy)**.** A randomized mechanism $\mathcal{M}$ is $\eta|_2-$differentially private if for all adjacent databases $d \sim d'$,

$$\Pr[\mathcal{M}(d) \in C] \leq 2^{\eta} \Pr[\mathcal{M}(d') \in C]$$

where probability is taken over the randomness of $\mathcal{M}$.

A very simple change of base proves the relationship between base-2 and base-e.

Lemma 2.2. *Any mechanism which is $\eta|_2$-differentially private is $\ln(2)\eta-$differentially private.*

Proof. $\Pr[\mathcal{M}(d) \in C] \leq 2^{\eta} \Pr[\mathcal{M}(d') \in C]$; $2^{\eta} = e^{\ln(2)\eta}$, thus $\Pr[\mathcal{M}(d) \in C] \leq e^{\ln(2)\eta} \Pr[\mathcal{M}(d') \in C]$. $\square$

We can also re-state the exponential mechanism in base-2.

Mechanism 1 ($|_2$ Exponential Mechanism)**.** Given a privacy parameter η, an outcome set O and a utility function $u : D \times O \rightarrow \mathbb{R}$ which maps (database, outcome) pairs to a real-valued utility, the $|_2$ exponential mechanism samples a single element from O based on the probability distribution

$$p(o) := \frac{2^{-\eta u(d,o)}}{\sum_{o \in O} 2^{-\eta u(d,o)}} \tag{1}$$

If $\Delta u \leq \alpha$, then the base-2 exponential mechanism is $2\alpha\eta|_2$-DP.

Note: we explicitly leave the sensitivity of the utility function (α) out of the distribution specification, as the implementation assumes that either η has been scaled to the appropriate sensitivity or the sensitivity is 1.

3 exponential_mechanism

		Type	Description
η	`eta`	`Eta`	Privacy parameter of the form $z \log_2(x/2^y)$ for `u32` x, y, z.
optimize	`arithmetic_config .optimize`	`bool`	Whether to optimize sampling.
optimize	`arithmetic_config .optimize`	`bool`	Whether to optimize sampling.
o_{max}	`max_outcomes`	`u32`	Maximum size of outcome space for exponential mechanism.
u_{min}	`utility_min`	`i64`	Minimum utility (maximum magnitude weight) value.
u_{max}	`utility_max`	`i64`	Maximum utility (minimum magnitude weight) value.
O	`outcomes`	`Vec<T>`	Vector of (generic type) outcomes.
U	`utilities`	`Vec<f64>`	The utilities for each outcome.
W	`weights`	`Vec<Float>`	The weight for each utility.
k	`arithmetic_config .retry_min`	`u32`	Minimum number of retries.

Table 1: Variables and Types for Exponential Mechanism

3.1 Functionality Claims and Proofs

We state the following proposition to formally characterize the behavior of Algorithm 1:

Theorem 3.1. *Given parameters u_{min} (`i64`), u_{max} (`i64`), o_{max} (`u32`), η (`Eta(x,y,z)` for `u32 x,y,z`), outcome set O (generic type `T`), and retry parameter k (`u32`) and utility function u (`Fn(&T)->f64`) determined independently of the database such that $\Delta u \leq 1$,[1] Algorithm 1 either*

(1) outputs an element from O sampled according to the probability of the Base-2 Exponential Mechanism (Mechanism 1, Equation 1) on the utility function $\mathsf{clamp}(u, u_{min}, u_{max})$ where

$$\mathsf{clamp}(u, A, B)(x, o) := \min(\max(A, u(x, o)), B)$$

or

(2) outputs an error if the precision is insufficient or inexact arithmetic occurs.

Proposition 3.2 (informal)**.** *Furthermore, except with probability 2^{-k}, the mechanism described in Algorithm 1 does not provide a useful timing (due to logic) or randomness side channel if precision is sufficient and no inexact arithmetic occurs. Timing information due to differences in floating point operation time may still be observable.*

We omit the proof of the proposition, and refer the user to the timing channel discussion in Section 4. This mechanism *should be considered vulnerable to timing channels based on floating point operation times*, but timing channels due to sampling logic are largely mitigated by appropriate choice of the `retry_min` (k) parameter. There are no documented timing channels related to precision determination.

[1] The choice of the function u must be independent of the database, but the utilities may depend on the database.

Proof of Theorem 3.1.

Proof. The proof of the theorem follows from the correctness of each of the components of the mechanism.

1. The mechanism first checks that the parameters are valid and determines the minimum working precision. The correctness of the precision follows from Theorem 5.4.
2. It then computes the utilities of each element of the outcome space, and applies randomized rounding. The privacy of randomized rounding in the exponential mechanism follows from Lemma 6.2.
3. Before computing the exponentiation base, $2^{-\eta}$, the mechanism begins monitoring inexact arithmetic.
4. The weight of each element in the outcome space $2^{-\eta U[i]}$ is then computed. If no inexact arithmetic is detected, then each weight has been computed exactly.
5. An index is then sampled via NormalizedSample, which either returns an error or samples correctly from the desired distribution as shown in Theorem 4.1.
6. Finally, the mechanism either returns the outcome sampled, or an error if inexact arithmetic was performed.

□

Notes

- Utility function. The utility function is specified as a function pointer that is assumed to have access to the underlying database. This prevents us from having any built-in assumptions about the form of the database or communication protocols, and instead allows utility function have significant flexibility. In general, it makes sense to construct utility functions from operations over the data set with well understood sensitivity, and common utility functions could also be provided. The utility function is assumed to be chosen independently from the database.
- **No verification of sensitivity of the utility function is provided by the mechanism.**
- The `ExponentialOptions` struct is a caller facing struct that includes mechanism options like whether or not to optimize. For completeness we include the struct definition below.
- The `ExponentialConfig` struct is an internal struct that takes care of validating the important parameters, and constructing the appropriate `ArithmeticConfig`. Also provides a wrapper around the `get_base` function provided by the privacy parameter `Eta` using the precision determined for the mechanism.
- The type of the outcome space `O` is generic to allow flexibility in the choice of outcome space.

3.2 Pseudocode

As the mechanism utilizes many helper structs and methods to ensure that parameters are properly specified, we give additional comments in the pseudocode to help match to the code.

Algorithm 1 Base-2 exponential mechanism

Inputs: *data independent parameters* u_{min} *(*`i64`*),* u_{max} *(*`i64`*),* $\eta(x, y, z)$ *(*`Eta {x,y,z}` *for* `u32 x, y, z`*),* o_{max} *(*`u32`*),* O *the set of outcomes,* k *(*`u32`*) a timing channel mitigation parameter, optimize a boolean indicating whether to use optimized sampling, empirical a boolean indicating whether to use empirically determined minimum precision, and utility function* u *(*`Fn(&T)->f64`*).*

Outputs: $o \in O$ *sampled according to the probability distribution of the base-2 exponential mechanism or an error.*

procedure B2ExponentialMechanism(u_{min} (`i64`), u_{max} (`i64`), o_{max} (`u32`), $\eta(x, y, z)$ (`Eta`), O (generic `T`), k (`u32`), optimize (`bool`), empirical (`bool`), u(`Fn(&T)->f64`))
- Check that $\eta(x, y, z)$ is a valid parameter choice ▷ `eta.check()?;`
- Confirm that $u_{min} < u_{max}$, $o_{max} > 0$. ▷ `exponential_config` construction confirms these parameters are valid.
- **if** empirical = false **then** ▷ Use analytical minimum precision, Lemma 5.3.
    - $p \leftarrow (\max(1, |u_{min}|) + \max(1, |u_{max}|))z(y + b_x) + o_{max}$
- **else**
    - $p \leftarrow$ GetEmpiricalPrecision($u_{min}, u_{max}, o_{max}, \eta$) ▷ Compute the empirical precision via Algorithm 3, Lemma 5.2
- Initialize an empty utility list U
- **for** $i \in \{0, 1, \ldots, |O| - 1\}$ **do** ▷ Rust Lines 200-214
    - $o \leftarrow O[i]$
    - Append `clamp`(RandomizedRound($u(o)$), u_{min}, u_{max}) to U ▷ See Lemma 6.2 for randomized rounding and Proposition 8.1 for clamping.
- Begin monitoring inexact arithmetic ▷ Rust Line 217
- $b \leftarrow 2^{-\eta}$ ▷ `exponential_config.get_base();` may return an error if insufficient precision.
- Initialize an empty weight list W
- **for** $i \in \{0, 1, \ldots, |O| - 1\}$ **do** ▷ Zero-indexing to match code.
    - $w_i \leftarrow b^{U[i]}$
    - Append w_i to W
- $i^* \leftarrow$ NormalizedSample(W, p, k) ▷ Rust Line 230
- **if** inexact arithmetic or i^* is error **then** ▷ `exit_exact_scope()` Rust Line 237
    - **return** error ▷ All `?` Rust lines may return error.
- $o^* \leftarrow O[i^*]$ ▷ Rust Line 234
- **return** o^*

Randomized rounding, see Section 6.2.

function RandomizedRound(x (`f64`))
- **return** $\begin{cases} \lfloor x \rfloor \text{ with probability } |x - \lceil x \rceil| \\ \lceil x \rceil \text{ with probability } |x - \lfloor x \rfloor| \end{cases}$

3.2.1 Rust Code

```
123 /// Implements the base-2 exponential mechanism.
124 /// Utility convention is to take `-utility(o)`, and `utility_min` is therefore the
    highest
125 /// possible weight/maximum probability outcome. This mechanism does not scale based
    on
126 /// the sensitivity of the utility function. For a utility function with sensitivity `
    alpha`,
127 /// the mechanism is `2*alpha*eta` base-2 DP, and `2*alpha*ln(2)*eta` base-e DP.
128 /// **The caller must ensure that `utility_min`, `utility_max`, `max_outcomes`
```

~~~rust
/// and `outcomes` are determined independently of the `utility` function and any private
/// data.**
///
/// ## Arguments
///   * `eta`: the base-2 privacy parameter
///   * `outcomes`: the set of outcomes the mechanism chooses from
///   * `utility`: utility function operating on elements of `outcomes`. `utility` does not
///                explicitly take a database input, and is expected to have a pointer to the database
///                or access to the private data needed to determine utilities.
///   * `utility_min`: the minimum utility permitted by the mechanism (highest possible weight)
///   * `utility_max`: the maximum utility permitted by the mechanism (lowest possible weight)
///   * `max_outcomes`: the maximum number of outcomes permitted by the mechanism
///   * `rng`: a random number generator
///
/// ## Returns
/// Returns a reference to an element in `outcomes` sampled according to the base-2 exponential
/// mechanism.
///
/// ## Known Timing Channels
/// **This mechanism has known timing channels.** Please see
/// [normalized_sample](../../utilities/exactarithmetic/fn.normalized_sample.html#known-timing-channels).
///
/// ## Errors
/// Returns Err if any of the parameters are configured incorrectly or if inexact arithmetic
/// occurs.
/// ## Example
/// ```
/// use b2dp::{exponential_mechanism, Eta, GeneratorOpenSSL, ExponentialOptions};
///
/// fn util_fn (x: &u32) -> f64 {
///     return ((*x as f64)-0.0).abs();
/// }
/// let eta = Eta::new(1,1,1).unwrap();
/// let utility_min = 0;
/// let utility_max = 10;
/// let max_outcomes = 10;
/// let rng = GeneratorOpenSSL {};
/// let options = ExponentialOptions {min_retries: 1, optimized_sample: true, empirical_precision: false};
/// let outcomes: Vec<u32> = (0..max_outcomes).collect();
/// let result = exponential_mechanism(eta, &outcomes, util_fn,
///                                     utility_min, utility_max,
///                                     max_outcomes,
///                                     rng, options);
/// ```
///
/// ## Exact Arithmetic
/// This function calls `enter_exact_scope()` and
/// `exit_exact_scope()`, and therefore clears the `mpfr::flags` and **does not preserve the
/// incoming flag state.**
pub fn exponential_mechanism<T, R: ThreadRandGen + Copy, F: Fn(&T)->f64>
        ( eta: Eta,
~~~

```
        outcomes: &Vec<T>,
        utility: F,
        utility_min: i64,
        utility_max: i64,
        max_outcomes: u32,
        rng: R,
        options: ExponentialOptions)
    -> Result<&T>
{
    // Check Parameters
    eta.check()?;
    if (max_outcomes as usize) < outcomes.len() {
        return Err("Number of outcomes exceeds max_outcomes.".into());
    }

    // Generate an ExponentialConfig
    let mut exponential_config = ExponentialConfig::new(eta,
                                                        utility_min,
                                                        utility_max,
                                                        max_outcomes,
                                                        options.empirical_precision,
                                                        options.min_retries)?;

    // Compute Utilities
    let mut utilities = Vec::new();
    for o in outcomes.iter() {
        let mut u = utility(o);
        // clamp the utility to the allowed range
        if u > exponential_config.utility_max as f64 {
            u = exponential_config.utility_max as f64;
        }
        else if u < exponential_config.utility_min as f64 {
            u = exponential_config.utility_min as f64;
        }
        utilities.push(randomized_round(u,
                                        & mut exponential_config.arithmetic_config,
                                        rng));
    }

    // Enter exact scope
    exponential_config.arithmetic_config.enter_exact_scope()?;

    // get the base
    let base = &exponential_config.get_base();

    // Generate weights vector
    let mut weights = Vec::new();
    for u in utilities.iter() {
        let w = exponential_config.arithmetic_config.get_float(base.pow(u));
        weights.push(w);
    }

    // Sample
    let sample_index = normalized_sample(&weights,
                                         & mut exponential_config.arithmetic_config,
                                         rng,
                                         options.optimized_sample)?;
    let sample = &outcomes[sample_index];

    // Exit exact scope
    exponential_config.arithmetic_config.exit_exact_scope()?;
```

```
    Ok(sample)
}
```

ExponentialOptions

```
/// The exponential mechanism optional parameters.
#[derive(Debug, Clone, Copy)]
pub struct ExponentialOptions {
    /// The minimum number of retries for timing channel prevention
    /// default: ‘1‘
    /// Minimum retries helps to mitigate timing channels in optimized
    /// sampling. The higher the number of retries, the less likely
    /// it is for an adversary to observe useful timing information.
    pub min_retries: u32,

    /// Whether to optimize sampling
    /// default: ‘false‘
    /// Optimized sampling exacerbates timing channels, and it’s not
    /// recommended for use in un-trusted settings.
    pub optimized_sample: bool,

    /// Whether to use empirical precision
    /// default: ‘false‘
    /// Determination of safe precision given utility bounds and outcome
    /// set size limits can be done analytically or empirically. Both
    /// are independent of the dataset. Using ‘empirical_precision = true‘
    /// determines the required precision via a set of test calculations.
    /// The timing overhead of these calculations is proportional to the outcome
    /// set size, and the overhead may outweigh any reduction in required
    /// precision.
    pub empirical_precision: bool,
}
impl Default for ExponentialOptions {
    /// Default options for the exponential mechanism
    /// ‘min_retries = 1‘, ‘optimized_sample = false‘, ‘empirical_precision = false‘
    fn default() -> ExponentialOptions
    {
        ExponentialOptions { min_retries: 1, optimized_sample: false,
   empirical_precision: false }
    }
}
```

ExponentialConfig

```
/// The exponential mechanism configuration. Includes all parameters
/// and information needed to derive the appropriate precision for the
/// mechanism.
#[derive(Debug)]
struct ExponentialConfig {
    /// The privacy parameter
    pub eta: Eta,
    /// The minimum utility (maximum weight) (signed)
    pub utility_min: i64,
    /// The maximum utility (minimum weight) (signed)
    pub utility_max: i64,
    /// The maximum size of the outcome space
    pub max_outcomes: u32,
```

```rust
    /// The arithmetic configuration
    arithmetic_config: ArithmeticConfig,
}

// Constructors
impl ExponentialConfig {
    /// Create a new context for the exponential mechanism.
    ///
    /// ## Arguments
    ///   * `eta`: the base-2 privacy parameter
    ///   * `utility_min`: the minimum utility permitted by the mechanism (highest possible weight)
    ///   * `utility_max`: the maximum utility permitted by the mechanism (lowest possible weight)
    ///   * `max_outcomes`: the maximum number of outcomes this instance exponential mechanism permits.
    ///
    /// ## Returns
    /// An `ExponentialConfig` from the specified parameters or an error.
    ///
    /// ## Errors
    /// Returns an error if any of the parameters are mis-specified, or if sufficient precision cannot
    /// be determined.
    pub fn new(eta: Eta,
               utility_min: i64,
               utility_max: i64,
               max_outcomes: u32,
               empirical_precision: bool,
               min_retries: u32)
            -> Result<ExponentialConfig>
    {
        // Parameter sanity checking
        if utility_min > utility_max {
            return Err("utility_min must be smaller than utility_max.".into());
        }
        if max_outcomes == 0 {
            return Err("Must provide a positive value for max_outcomes.".into());
        }

        let arithmetic_config =  ArithmeticConfig::for_exponential(&eta,
                                                                    utility_min,
                                                                    utility_max,
                                                                    max_outcomes,
                                                                    empirical_precision,
                                                                    min_retries)?;

        // Construct the configuration with the precision we determined above
        let config = ExponentialConfig {
            eta,
            utility_min,
            utility_max,
            max_outcomes,
            arithmetic_config
        };
        Ok(config)
    }

    /// Wrapper function for `Eta::get_base`. Returns
    /// `eta.get_base()` using the precision specified by
```

```rust
    /// `self.arithmetic_config`.
    pub fn get_base(&self) -> Float {
        self.eta.get_base(self.arithmetic_config.precision).unwrap()
    }
}
```

4 normalized_sample

		Type	Description
η	`eta`	`Eta`	Privacy parameter of the form $z \log_2(x/2^y)$ for `u32` x, y, z.
p	`arithmetic_config .precision`	`u32`	Precision. Recall that `arithmetic_config .get_float` and `.get_rand_float` return `Float` with precision `arithmetic_config.precision`.
weights	`weights`	`Vec<Float>`	The vector of weights for sampling.
total_weight	`total_weight`	`Float`	The sum of the `weights` vector.
retry_min	`arithmetic_config .retry_min`	`u32`	Minimum number of retries.
optimize		`bool`	Whether to optimize sampling.
t	`t`	`Float`	Intermediate value
s	`s`	`Float`	Intermediate value (a random Float in $[0, 2^k)$ where $2^k \geq total\_weight$)
cweight	`cumulative_weight`	`Float`	Intermediate value (running cumulative weight)
index	`index`	`Option<usize>`	Intermediate value (the index to return)

Table 2: Variables and Types for Normalized Sampling

4.1 Functionality Claims and Proofs

Normalized sample takes in a set of weights, and samples an index based on the normalized weights.

There are two types of properties we care about: correctness of sampling (the main theorem) and timing channel properties (omitted, please see full paper).

Theorem 4.1. NormalizedSample *given a set of weights (*`Vec<Float>`*) and a precision p (*`u32`*) returns an index (*`usize`*) (within the bounds of the weights list) according to $Pr(i) := \frac{weights[i]}{\sum weights}$, or returns an error if precision is insufficient to do so. That is $\forall i \in [|weights|]$, $\Pr[return\ value = i \mid return\ value\ is\ not\ an\ error] = \frac{weights[i]}{\sum weights}$.*

Proof. The proof consists of three parts: first, correctness of the distribution assuming sufficient precision; second, sufficiency of precision; and third, identification of error conditions.

Correctness. Notice that (assuming infinite precision) this procedure amounts to partitioning the range $[0, total\_weight)$, between the elements of *weights* according to their weight, i.e., element i is assigned range $[weights[i-1], weights[i])$, sampling a value s in $[0, total\_weight)$ and choosing an element in *weights* based on which partition s lands in. To see the correctness of the sampling procedure, observe that in each iteration of the **while** loop (Algorithm 2 Line 15), a value is sampled in the range $[0, 2^k)$, where $k = \arg\min_{k \in \mathbb{N}}\{2^k \geq total\_weight\}$, and either discarded if $s \geq total\_weight$ or kept. The values that are kept are therefore sampled uniformly in $[0, total\_weight)$. Thus, the probability that any given element is sampled is equivalent to the probability that a random value $s \in [0, total\_weight)$ falls into its assigned range of $[0, total\_weight)$, thus, each index is sampled with probability $\frac{weights[i]}{total\_weight}$, which is equivalent to the exponential mechanism.

Sufficient precision. Let c_i denote the value of *cweight* for the i^{th} iteration of the loop in line 23. Notice that p bits are sufficient to express any c_i for $i \in 0, \ldots, |weights| - 1$. Imagine that an oracle agrees to read out a random value in $[0, total\_weight)$ with infinite bits of precision. After hearing p bits, we have sufficient information to choose a single value in *weights*, and hearing any more bits cannot change our choice. The sampling procedure for s in line 16 is equivalent to taking the first p bits from the oracle. This follows from observing that at most one element can "claim" any range $[a2^{-p}, (a+1)2^{-p})$ as all combinations of *weights* can be expressed in p bits of precision. Thus, p bits of precision are sufficient to simulate the infinite precision procedure.

Error conditions. Suppose that sufficient precision is not available, i.e., p is not sufficient to express all of the cumulative weights. There are two possible cases, either (1) the value sampled covered a region containing $[s, s + 2^{k-p})$, and hearing any more bits of s^* wouldn't change the outcome or (2) an error is returned. Thus, either the sample is the sample that would have been drawn given infinite precision, or an error is returned. □

Notes:

- We disallow weights of zero to prevent ambiguity. If there is one zero weight and many positive weights, this might be interpreted as the zero-weight element having probability zero of being sampled. However, if all weights are zero, should an element be returned at random (as all weights are equal?) or should we return an error. To simplify the interface, we require positive weights.
- We take $|weights|$ to be the number of elements in the vector *weights*. **Aside:** I'm happy to adopt a different convention here, we could use *weights.length*().
- The precision used in the code comes from the `ArithmeticConfig`. All methods for changing the precision in the `ArithmeticConfig` ensure that the precision is within the allowed range on the system the code is operating on (i.e., that it's a positive integer smaller than the maximum precision). (See Section 5.)
- The value of s sampled in Line 16 is implemented via the `get_rand_float` function from the Arithmetic Config, which produces a Float of the precision specified in the config in the range $[0, 1)$ with each bit drawn uniformly at random.
- Timing channels are somewhat mitigated by increasing the minimum number of loop retries, but this does not address timing channels due to differences in floating point operations. Users should not consider this code to be entirely timing channel safe, but they may increase the `retry_min` parameter to reduce the likelihood of an adversary observing useful timing information.
- Inexact arithmetic versus insufficient precision. As written, it should not be possible for calls to `normalized_sample` to result in an insufficient precision error unless the working precision of the `ArithmeticConfig` changes at run-time. In general, using insufficient precision for this method will result in an inexact arithmetic error instead (as constructing a `Float` which results in truncation counts as "inexact arithmetic"). Please see `test_insufficient_sampling_precision()` for example error conditions.

4.2 Pseudocode

Pseudocode for the method is included in Algorithm 2 below.

Algorithm 2 Normalized Sample

Inputs: *An array of positive weights (*`Vec<Float>`*), a precision p (*`u32`*) which is a positive integer value less than the maximum system precision, a source of randomness rng, a boolean optimization parameter, and a minimum number of retries retry_min (*`u32`*).*

Outputs: *An index sampled according to the normalized weights, i.e.,* $\Pr[output\ i] := \frac{weights[i]}{\sum weights}$.

procedure NORMALIZED_SAMPLE($weights$, p, rng, $optimize$, $retry\_min$)
    $total\_weight \leftarrow \sum_{i \in [|weights|]} weights[i]$ ▷ Total weight
    if $total\_weight = 0$ **then**
        return Error
    for $w \in weights$ **do**
        if $w = 0$ **then**
            $zweight \leftarrow 1$ ▷ Error if there is a zero weight.
            if $optimize = 1$ **then**
                return Error
    if $zweight = 1$ **then** ▷ Return after full loop to prevent timing channel.
        return Error
    $retries \leftarrow 0$
    $k \leftarrow$ GET_POWER_BOUND($total\_weight$) ▷ $\arg\min_{k \in \mathbb{N}} \{2^k \geq total\_weight\}$
    $t \leftarrow total\_weight + 1$
    while $t \geq total\_weight$ or $retries <$ `retry_min` **do** ▷ Rust lines 145-156.
        $s \sim \mathbf{Unif}([0, 2^p)) * 2^{-p}$ ▷ s is a uniformly random p bit value between $[0, 1)$, see `ArithmeticConfig::get_rand_float()`.
        $s \leftarrow s * 2^k$ ▷ s is scaled to between $[0, 2^k)$
        if $s < total\_weight$ **then**
            $t \leftarrow s$
        $retries \leftarrow retries + 1$
    $cweight \leftarrow 0$
    $index \leftarrow None$
    for $i = \{0, \ldots, |weights| - 1\}$ **do** ▷ Zero-indexing to match code
        $cweight \leftarrow cweight + weights[i]$
        if $cweight > t$ **then**
            if $index$ is $None$ **then**
                Check that $cweight + weights[i+1]$ is not in between $cweight$ and the next largest value that can be expressed with the given p.
                if Check fails **then**
                    return insufficient precision error
                $index \leftarrow i$
                if $optimize$ **then**
                    return $index$
    return $index$

4.2.1 Rust Code

Source File: `exactarithmetic.rs`.

```rust
/// Normalized Weighted Sampling
/// Returns the index of the element sampled according to the weights provided.
/// Uses optimized sampling if `optimize` set to true. Setting `optimize` to true
/// exacerbates timing channels.
/// ## Arguments
```

```
///   * `weights`: the set of weights to use for sampling; all weights must be positive,
///                 zero-weight elements are not permitted.
///   * `arithmetic_config`: the arithmetic config specifying precision
///   * `rng`: source of randomness.
///   * `optimize`: whether to optimize sampling, introducing a timing channel and an error condition
///                  side channel.
/// ## Returns
/// Returns an index of an element sampled according to the weights provided. If the precision
/// of the provided ArithmeticConfig is insufficient for sampling, the method returns an error.
/// Note that errors are **not** returned on inexact arithmetic, and the caller is responsible
/// for calling `enter_exact_scope()` and  `exit_exact_scope()` to monitor inexact arithmetic.
///
///
/// ## Known Timing Channels
/// This method has known timing channels. They result from:
/// (1) Generating a random value in [0,2^k] and
/// (2) (In optimized sampling only) To determine the index corresponding to the random value,
/// the method iterates through cumulative weights
/// and terminates the loop when the index is found and
/// (3) (In optimized sampling only) Checking for zero weights
/// These can be exploited in several ways:
///   * **Rejection probability:** if the adversary can control the total weight of the utilities
///     such that the probability of rejection in the random index generation stage changes,
///     the time needed for sampling will vary between adjacent databases. The difference in time
///     will depend on the speed of random number generation. By default, ArithmeticConfig sets the
///     minimum retries to 1. To reduce the probability that this timing channel is accessible to an
///     adversary, the minimum number of retries can be increased via `ArithmeticConfig::set_retries`.
///   * **Optimized sampling:**
///     * **Ordering of weights:** if the adversary can change the ordering of the weights such
///       that the largest weights (most probable) weights are first under a certain condition,
///       and the largest weights are last if that condition doesn't hold, then the adversary
///       can use the time of normalized_sample to guess whether the condition holds.
///     * **Size of weights:** if the adversary can change the size of the weights such that if
///       a certain condition holds, the weight is more concentrated and if not the weight is less
///       concentrated, then the adversary can use the time taken by normalized_sample as a signal
///       for whether the condition holds.
///     * **Zero weight:** optimized sampling also rejects immediately if a zero weight is encountered.
///       If the adversary can inject a zero weight at a particular position in the weights depending on
///       a private condition, they can use the time it takes to return an error as a timing channel.
```

```
///
/// The timing channels for optimized sampling could be somewhat (but not completely) mitigated by
/// shuffling the weights prior to calling `normalized_sample`.
/// ### Exact Arithmetic
/// `normalized_sample` does not explicitly call `enter_exact_scope()` or
/// `exit_exact_scope()`, and therefore preserves any `mpfr::flags` that
/// are set before the function is called.

pub fn normalized_sample<R: ThreadRandGen>(
                                            weights: &Vec<Float>,
                                            arithmetic_config: &mut ArithmeticConfig,
                                            mut rng: R,
                                            optimize:bool,
                                        ) -> Result<usize, &'static str>
{
    // Compute the total weight
    let total_weight = arithmetic_config.get_float(Float::sum(weights.iter()));
    if total_weight == 0 { return Err("Total weight zero. Weights must be positive."); }
    let mut zero_weight: Option<()> = None;

    // Iterate through all weights to test to prevent timing channel,
    // unless `optimize = true`.
    for w in weights.iter() {
        if w.is_zero() {
            zero_weight = Some(());
            if optimize  { return Err("All weights must be positive."); }
        }
    }

    if zero_weight.is_some() {return Err("All weights must be positive.");}
    // Determine smallest `k` such that `2^k >= total_weight`
    let k = get_power_bound(&total_weight, arithmetic_config);

    let mut t = arithmetic_config.get_float(&total_weight);
    let mut retries = 0;

    t += 1; // ensure that the initial `t` is larger than `total_weight`.
    while t >= total_weight || retries < arithmetic_config.retry_min {
        let mut s = arithmetic_config.get_rand_float(&mut rng);
        // Multiply by 2^k to scale
        // Note: Float::i_exp(a,b) returns a*2^b
        let two_pow_k = arithmetic_config.get_float(Float::i_exp(1, k));
        s = s * two_pow_k;
        // Assign to t if in bounds
        if s < total_weight {
            t = arithmetic_config.get_float(&s);
        }
        retries += 1; // increment retries
    }
    if t >= total_weight {return Err("Failed to produce t");}
    let mut cumulative_weight = arithmetic_config.get_float(0);
    let mut index: Option<usize> = None;

    // Iterate through the weights until the cumulative weight is greater than or equal to `t`
    for i in 0..weights.len() {
        let next_weight = arithmetic_config.get_float(&weights[i]);
        cumulative_weight += next_weight;
        if cumulative_weight > t {
```

```
            // This is the index to return
            if index.is_none() {
                // Check sufficient precision
                let mut next_highest = arithmetic_config.get_float(&t);
                next_highest.next_up();
                if i < weights.len() - 1 {
                    let next_weight = arithmetic_config.get_float(&weights[i+1]);
                    let mut cumulative_next = arithmetic_config.get_float(&cumulative_weight);
                    cumulative_next = cumulative_next + next_weight;
                    if cumulative_next < next_highest {
                        return Err("Sampling precision insufficient");
                    }
                }
                index = Some(i);
                if optimize {
                         return Ok(i);
                }
            }
        }

    }

    if index.is_some() { return Ok(index.unwrap()); }

    // Return an error if we are unable to sample
    // Caller can choose an index at random if needed
    Err("Unable to sample.")
}
```

5 ArithmeticConfig

5.1 Struct

The `ArithmeticConfig` struct is essentially a wrapper around all of the inexact arithmetic monitoring and precision logic. It provides several useful helper functions including `get_float` and `get_rand_float` which return `Float`s with precision inherited from the `ArithmeticConfig`. In general, if you want to enforce exact arithmetic, it's best to construct all `Float`s used via these methods to ensure that they have the correct precision. (We omit detailed proof for these helper methods, as the code is self-explanatory.)

5.1.1 Rust Code

```
/// The exact arithmetic configuration. Includes the precision of all
/// mechanism arithmetic and status bits indicating if any inexact
/// arithmetic has been performed.
/// The ArithmeticConfig implementation encapsulates all `unsafe` calls to
/// `mpfr`.
#[derive(Debug)]
pub struct ArithmeticConfig {
    /// The required precision (computed based on other parameters)
    pub precision: u32,
    /// Whether an inexact operation has been performed in the scope of
    /// this config
    pub inexact_arithmetic: bool,
    /// Whether the code is currently in an exact scope
    exact_scope: bool,
    /// The number of retries for timing channel prevention
```

```
    /// default is 1.
    retry_min: u32,
}
```

5.2 Exact Arithmetic Monitoring

For exact arithmetic monitoring, the caller should use `enter_exact_scope` to clear the current flags and start monitoring exact arithmetic and `exit_exact_scope` when the exact arithmetic is completed and they want to confirm that no inexact arithmetic was performed.

Proposition 5.1. *If inexact arithmetic is performed on a* `Float` *including addition, multiplication, exponentiation, etc resulting in overflow, underflow, inexact result*[2]*, between a call of* `enter_exact_scope` *and* `exit_exact_scope`*, then* `exit_exact_scope` *will return an error as long as no other code clears the* `mpfr::flags`*.*

We state this as a proposition as correctness is self-explanatory. However, it is critical to note that any other code which interacts with the `mpfr::flags` at runtime breaks this assumption. In particular, **we have not tested or designed this code to work in a multi-threaded environment (i.e., multiple `ArithmeticConfigs` concurrently entering and exiting exact scopes).**

5.2.1 Rust Code

```
    /// Enter exact arithmetic scope.
    /// This method clears `mpfr` flags if not currently in an `exact_scope`.
    /// # Returns
    ///   * `OK(())` if the scope is successfully entered
    ///   * `Err` if the scope is alread invalid
    pub fn enter_exact_scope(&mut self) -> Result<(), &'static str> {
        if self.is_valid() == false {
            // inexact arithmetic has already occurred
            return Err("ArithmeticConfiguration invalid.");
        }
        if !self.exact_scope {
            unsafe {
                mpfr::clear_flags();
            }
            // set the exact_scope flag
            self.exact_scope = true;
        }

        return ArithmeticConfig::check_mpfr_flags();
    }

    /// Exit the exact arithmetic scope.
    /// **Must be called after any arithmetic operations are performed which should be exact.**
    /// **Must be paired with `enter_exact_scope` to ensure that flags aren't misinterpreted.**
    /// This method checks the `mpfr` flag state, and returns whether
    /// the scope is still valid. Also sets the `inexact` property.
    /// This method does **not** reset the `mpfr` flags.
    ///
    /// ## Returns
    ///   * `OK(())` if the configuration reports than no inexact arithmetic was performed
    ///   * `Err` if the configuration is invalid (inexact arithmetic performed)
```

[2] See `https://tspiteri.gitlab.io/gmp-mpfr-sys/mpfr/MPFR-Interface.html#Exception-Related-Functions` for the complete list of conditions

```
    pub fn exit_exact_scope(&mut self) -> Result<(), &'static str> {
        if self.is_valid() == false {
            // Error has already occurred
            return Err("ArithmeticConfiguration invalid.");
        }
        if self.exact_scope == false {
            return Err("Not in exact scope.");
        }
```

5.3 Precision Determination for Exponential Mechanism

Please refer to Table 3 for types and variable names.

There are two ways of determining the precision needed for the exponential mechanism: empirically and analytically (worst-case). We need precision sufficient to compute any combination (subset sum) of the weights $2^{-\eta u(d,o)}$ for any set of utilities within the specified clamping bounds in order to sample from the desired distribution. The first technique is to compute a set of "worst case" sums, and increase the precision until it is sufficient to compute the sums exactly. The second technique (see Lemma 5.3) is to compute the worst-case minimum precision required analytically. It is discussed in the next subsection.

The worst-case empirical precision is computed by taking the set of "worst-case" sums, i.e., the maximum total weight plus the weight with the largest precision requirement after the decimal. This corresponds to the highest precision required to compute a subset sum. Algorithm 3 outlines the procedure in detail.

		Type	Description
η	`eta`	`Eta`	Privacy parameter of the form $z \log_2(x/2^y)$ for `u32` x, y, z such that $x/2^y < 1$.
p	`p`	`u32`	Precision.
o_{max}	`max_outcomes`	`u32`	Maximum size of outcome space for exponential mechanism.
u_{min}	`utility_min`	`i64`	Minimum utility (maximum magnitude weight) value.
u_{max}	`utility_max`	`i64`	Maximum utility (minimum magnitude weight) value.
w_{max}	`max_weight`	`Float`	Maximum possible subset sum of weights
$combsum$	`_combination`	`Float`	Intermediate value

Table 3: Variables and Types for Precision Determination

Lemma 5.2. *Algorithm 3 either returns a precision sufficient to exactly compute any subset sum of at most o_{max} (`u32`) integer utilities in the range $[u_{min}, u_{max}]$ (where u_{min}, u_{max} are `i64`) with privacy parameter η, i.e. the sum of at most o_{max} utilities of the form $2^{-\eta u}$ for $u \in [u_{min}, u_{max}] \cap \mathbb{N}$, or returns in error if such a precision cannot be determined.*

Proof. The worst-case empirical procedure (Algorithm 3) computes every hypothetical worst case, and reports the required precision. More concretely, given a bound on the number of outcomes (o_{max}) and a range of utilities ($[u_{min}, u_{max}]$) with maximum weight $w_{max} = 2^{-\eta u_{min}}$, it suffices to ensure that we have sufficient precision to calculate (1) each weight independently, i.e., $2^{-\eta u}$ for integer $u \in [u_{min}, u_{max}]$ and (2) the maximum possible sum of the weights plus the weight with highest fractional precision (w_*), i.e. $\lceil o_{max} w_{max} \rceil + w_*$. (Note that w_* is not necessarily the smallest weight.) This follows from observing that the maximum number of bits required for the mantissa will be dictated by the largest possible sum of weights and the highest fractional precision needed to express any individual weight. Thus if the precision is sufficient to express w_i and $\lceil o_{max} w_{max} \rceil + w_i$ for all $i \in [u_{min}, u_{max}]$ where $w_i = 2^{-\eta i}$, then it is sufficient to compute the sum of any valid subset of weights. Algorithm 3 implements this procedure by first determining the minimum precision for the "base" of $2^{-\eta}$, and attempts to compute the maximum sum of $maxsum = o_{max} 2^{-\eta u_{min}}$ and each subset sum $\lceil maxsum \rceil + 2^{-\eta u}$ for each integer $u \in [u_{max}, u_{min}] \cap \mathbb{N}$. $\square$

5.3.1 Pseudocode

Algorithm 3 Minimum precision determination

Inputs: u_{min} (`i64`), *the minimum utility,* u_{max} (`i64`), *the maximum utility,* o_{max} (`u32`) *the maximum number of outcomes, the privacy parameter* η (`Eta {x,y,z}` *for* `u32` *x, y, z).*
Outputs: p (`u32`), *a sufficient precision no more than twice the size of the minimum precision to successfully run the base-2 exponential mechanism.*

procedure GetEmpiricalPrecision(u_{min} (`i64`), u_{max} (`i64`), o_{max} (`u32`), η (`Eta`))
  $p \leftarrow$ system default
  while inexact arithmetic for $2^{-\eta}$ **do** ▷ Make sure we can compute $2^{-\eta}$ exactly.
    $p \leftarrow 2p$
    if $p \geq$ system maximum **then return** error
  while CheckPrecision($u_{min}, u_{max}, o_{max}, \eta, p$) fails **do**
    $p \leftarrow 2p$ ▷ $2p$ can be changed to a fixed increment if preferred.
  return p

Computes the worst-case sums and returns failure on inexact arithmetic.

function CheckPrecision(u_{min} (`i64`), u_{max} (`i64`), o_{max} (`u32`), η (`Eta`), p (`u32`))
  Set the precision to p
  Begin monitoring inexact arithmetic
  $w_{max} \leftarrow 2^{-\eta u_{min}}$
  $maxsum \leftarrow w_{max} * o_{max}$
  for $u \in [u_{min}, u_{max}]$ **do**
    $combsum \leftarrow 2^{-\eta u} + \lceil maxsum \rceil$
  Stop monitoring inexact arithmetic
  if inexact arithmetic **then return** failure
  else return success

5.3.2 Rust Code

Source File: `exactarithmetic.rs`.

```rust


    /// Computes the precision necessary for the exponential mechanism
    /// by computing a set of worst-case sums.
    /// Does **not** preserve the state of the incoming `mpfr::flags`.
    unsafe fn get_empirical_precision(eta: &Eta,
                                      utility_min: i64,
                                      utility_max: i64,
                                      max_outcomes: u32,
                                      max_precision: u32,)
                                      -> Result<u32,&'static str>
    {

        // Clear the flags
        mpfr::clear_flags();
        let mut p = mpfr::get_default_prec() as u32;
        // Get the base with the default precision
        let mut _base_test = eta.get_base(p);

```

```
        while ArithmeticConfig::check_mpfr_flags().is_err() {
            p *= 2;
            // Clear the flags
            mpfr::clear_flags();
            // Check if the precision has exceeded the maximum allowed
            if p > max_precision {
                return Err("Maximum precision exceeded.");
            }
            _base_test = eta.get_base(p);
        }
        // Check that we can compute the base with the current precision.
        mpfr::clear_flags();
        let base_result = eta.get_base(p);

        ArithmeticConfig::check_mpfr_flags()?;
        let base = &base_result.unwrap();

        // Loop until we can successfully evaluate the test function.
        mpfr::clear_flags(); // clear flags
        mpfr::set_inexflag(); // set the inexact flag
        let mut opt_p: Option<u32> = None;
        while ArithmeticConfig::check_mpfr_flags().is_err() {
            mpfr::clear_flags();
            // Increase the precision and update the base to the new precision
            // Only double if we haven't tried at this precision yet.
            if opt_p.is_none() { opt_p = Some(p); }
            else { p *= 2; }
            let new_base = &Float::with_val(p, base);
            // Check if the precision has exceeded the maximum allowed
            if p > max_precision {
                return Err("Maximum precision exceeded.");
            }

            for u in utility_min..(utility_max+1) {
                let max_weight = Float::with_val(p, new_base.pow(utility_min)).ceil();
```

5.4 Constructor for Exponential Mechanism

The primary consideration for correctness of the constructor for the `ArithmeticConfig` for the exponential mechanism is that it chooses the appropriate precision. Correctly instantiating the flags for inexact arithmetic, retry_min, etc, are just assignments.

We first prove a lemma concerning the analytical minimum precision required for the exponential mechanism. *Please refer to Table 3 for variable types.*

Lemma 5.3. *Given a range of utilities u_{min} (`i64`) to u_{max} (`i64`), a maximum number of outcomes o_{max} (`u32`) and a privacy parameter $\eta = -z\log_2(\frac{x}{2^y})$ such that x, y, z are positive integers and $\frac{x}{2^y} \leq 1$, the sum of any subset of at most o_{max} weights of the form $2^{-\eta u}$ computed from integer utilities within the range $[u_{min}, u_{max}]$ requires at most $(\max(1, |u_{min}|) + \max(1, |u_{max}|))z(y + b_x) + o_{max}$ bits of precision. Where b_x is the number of bits required to write the value x in binary.*

Proof. Given a binary number q and an integer n, q^n can be written with $\max(1, b_q|n|)$ bits of precision. Writing $2^{-\eta u}$ requires no more than $\max(1, |u|)z(y + b_x)$ bits of precision. Thus to write the largest weight, corresponding to u_{min}, we require no more than $\max(1, |u_{min}|)z(y+b_x)$ bits. The largest possible value any combination of weights could take on is $o_{max}2^{-\eta u_{min}}$, and adding $2^{-\eta u_{min}}$ to itself o_{max} times requires an extra o_{max} bits of precision, as each addition increases the precision by at most one bit. Thus, the largest possible combination requires $o_{max} + \max(1, |u_{min}|)z(y + b_x)$ bits. This corresponds to the largest possible number of bits needed before the decimal. To compute the largest possible number of bits needed after the decimal, we consider the smallest possible weight, $2^{-\eta u_{max}}$, which will require $|u_{max}|z(y+b_x)$ bits. Thus in the worst case, we require maximum precision on both sides of the decimal, yielding the desired maximum bound of $(\max(1, |u_{min}|) + \max(1, |u_{max}|))z(y + b_x) + o_{max}$. □

The pseudocode for analytical empirical determination is omitted, as it is fully described in the expressions above. The code implementing this expression is included in Lines 324-331 of the code below.

Theorem 5.4. *The constructor* `for_exponential` *instantiates an* `ArithmeticConfig` *with sufficient precision to execute the exponential mechanism exactly.*

Proof. The proof of the theorem follows from the either Lemma 5.3 or Lemma 5.2 depending on whether the `empirical_precision` parameter is set to `True` or `False`. □

5.4.1 Rust Code

```
280             }
281
282         }
283
284         ArithmeticConfig::check_mpfr_flags()?;
285         Ok(p)
286     }
287
288
289     /// Initialize an ArithmeticConfig for the base-2 exponential mechanism.
290     /// This method empirically determines the precision required to compute a linear
291     /// combination of at most `max_outcomes` weights in the provided utility range.
292     /// Note that the precision to create Floats in rug/mpfr is given as a `u32`, but the
293     /// sizes (min, max, etc) of precision returned (e.g. `mpfr::PREC_MAX`) are `i64`.
294     /// We handle this by explicitly checking that `mpfr::PREC_MAX` does not exceed the
295     /// maximum value for a `u32` (this should never happen, but we check anyway).
296     ///
297     /// ## Arguments
298     ///   * `eta`: the base-2 privacy parameter
299     ///   * `utility_min`: the minimum utility permitted by the mechanism (highest possible weight)
300     ///   * `utility_max`: the maximum utility permitted by the mechanism (lowest possible weight)
301     ///   * `max_outcomes`: the maximum number of outcomes permitted by this instance of the exponential
302     ///                      mechanism.
303     ///
304     /// ## Returns
305     /// Returns an ArithmeticConfig with sufficient precision to carry out the operations for the
306     /// exponential mechanism with the given parameters.
307     ///
308     /// ## Errors
309     /// Returns an error if sufficient precision cannot be determined.
310     pub fn for_exponential(
311                             eta: &Eta,
312                             utility_min: i64,
313                             utility_max: i64,
314                             max_outcomes: u32,
315                             empirical_precision: bool,
316                             min_retries: u32,
```

```
                        ) -> Result<ArithmeticConfig, &'static str>
    {
        let p: u32;

        unsafe {
            // Clear the flags
            mpfr::clear_flags();

            // Check that the maximum precision does not exceed the maximum value of a
            // u32. Precision for Float::with_val(precision: u32, val) requires a u32.
            let mut max_precision = u32::max_value();
            if mpfr::PREC_MAX < max_precision as i64 {
                max_precision = mpfr::PREC_MAX as u32;
            }

            if !empirical_precision{
                let mut bx =  (eta.x as f32).log2().ceil() as u32;
                if bx < 1 { bx = 1;}
                let mut um = utility_max.abs();
                if um < 1 { um = 1; }
                if utility_min.abs() < 1 { um += 1; }
                else { um += utility_min.abs(); }
                p = (um as u32)*(eta.z*(eta.y+bx)) as u32 + max_outcomes;
                if p > max_precision {return Err("Maximum precision exceeded."); }
            }
            else
            {
                p = ArithmeticConfig::get_empirical_precision(&eta, utility_min,
    utility_max, max_outcomes, max_precision)?;
            }
        } // end unsafe block
```

6 Exact Arithmetic Helper Methods

Source File: `exactarithmetic.rs`.

6.1 get_power_bound

Lemma 6.1. *`get_power_bound` computes the smallest value of k (`i32`) such that $2^k \geq total\_weight$ given the positive argument $total\_weight$ (`Float`), i.e., $\arg\min_{k \in \mathbb{N}}\{2^k \geq total\_weight\}$. If $total\_weight <= 0$, returns 0.*

(See code below for pseudocode).

Proof. The computation proceeds in two cases depending on whether $total\_weight$ is (A) greater than or (B) less than or equal to 1.

Case A. Initialize $k = 0$, then compute 2^k and increment k until $2^k \geq total\_weight$, which necessarily chooses the smallest value of k such that $2^k \geq total\_weight$.

Case B. Another way to compute $\arg\min_{k \in \mathbb{N}_{\geq 0}}\{2^k \geq total\_weight\}$ is to take $k = -t$ for

$$\arg\max_{t \in \mathbb{N}_{\geq 0}}\{2^t total\_weight < 1\}.$$

Initialize $k = 0$. Take $w = total\_weight$. While $w <= 1$, multiply w by 2 and decrement k by 1. Notice that the first iteration of the loop when $w <= 1$ corresponds to the point when k has been sufficiently decremented such that $2^{-k} total\_weight <= 1$, thus $k + 1$ is the smallest k such that $2^k \geq total\_weight$. $\square$

6.1.1 Rust Code

```rust
/// Determine smallest `k` such that `2^k >= total_weight`.
/// Returns zero if `total_weight` <= 0.
fn get_power_bound(total_weight: &Float,
                   arithmetic_config: &mut ArithmeticConfig)
  -> i32
{
    let mut k: i32 = 0;
    if *total_weight <= 0 { return 0; }
    if *total_weight > 1 {
        // increase `k` until `2^k >= total_weight`.
        let mut two_exp_k = Float::i_pow_u(2, k as u32);
        while arithmetic_config.get_float(two_exp_k) < *total_weight {
            k += 1;
            two_exp_k = Float::i_pow_u(2, k as u32);
        }
    } else {
        let mut w = arithmetic_config.get_float(total_weight);
        while w <= 1 {
            k -= 1;
            w *= 2;
        }
        k += 1;
    }
    k
}
```

6.2 `randomized_round`

`randomized_round` chooses $\lfloor x \rfloor$ or $\lceil x \rceil$ by drawing a random value ρ with precision determined by `arithmetic_config` and rounding down if $\rho > x - \lfloor x \rfloor$ and rounding up otherwise. Please see rust code for pseudocode and to confirm correctness.

6.2.1 Privacy of Randomized Rounding

Aside: We may want to rewrite this explicitly as base 2, but the logic is identical.

Lemma 6.2 (Privacy of arbitrary precision randomized rounding)**.** *Given an implementation of a utility function* $u : O \to$ `Float` *which guarantees that for any pair of adjacent databases* $|u(d', o) - u(d, o)| \leq \alpha$ *for integer* α *as implemented*[3]*, the exponential mechanism with a randomized rounding function of arbitrary precision is* $2\alpha\varepsilon-DP$*.*

Proof. Suppose to implement randomized rounding that we draw a number s uniformly at random from $[0, 1)$, and round up if $s \leq |u(d, o) - \lfloor u(d, o) \rfloor|$ (and otherwise round down). Fix a particular choice of s. Consider a pair of adjacent databases d and d' such that $u(d', o) > u(d, o)$. Notice that it is impossible for the rounding procedure to result in a difference in composed utility of more than α. This follows from observing that there are two cases: either $\lfloor u(d', o) \rfloor - \lfloor u(d, o) \rfloor < \alpha$, in which case any rounding results in difference at most α, or $\lfloor u(d', o) \rfloor - \lfloor u(d, o) \rfloor = \alpha$.

If $\lfloor u(d', o) \rfloor - \lfloor u(d, o) \rfloor = \alpha$, then $\lfloor u(d', o) \rfloor - \lfloor u(d, o) \rfloor \geq u(d', o) - u(d, o)$, and thus $u(d', o) - \lfloor u(d', o) \rfloor \leq u(d, o) - \lfloor u(d, o) \rfloor$. Therefore s is in one of three regions:

1. $s \in [0, u(d', o) - \lfloor u(d', o) \rfloor]$, which results in both values rounded up,

[3] By "as implemented" we mean that the implementation of u has sensitivity $\leq \alpha$. If inexact implementation of u results in increased sensitivity, then this must be taken into account.

2. $s \in (u(d', o) - \lfloor u(d', o) \rfloor, u(d, o) - \lfloor u(d, o) \rfloor]$, which results in $u(d', o)$ rounded down and $u(d, o)$ rounded up, or

3. $s \in (u(d, o) - \lfloor u(d, o) \rfloor, 1)$, which results in both rounded down.

Thus, for any s rounding never results in a difference between $u(d, o)$ and $u(d', o)$ greater than α. (The symmetric argument follows for any o such that $u(d, o) > u(d', o)$.) Take the utility function $u_S := \rho(u(d, o))$ to be the utility function with fixed randomness S, i.e., the set of s used for each rounding decision. From the above, $\Delta u_S \leq \alpha$. Thus, the exponential mechanism with utility function u_S is $2\alpha\varepsilon$−DP. Write $p_S(o)$ for the probability that the exponential mechanism with utility function u_S outputs the element o. Taking $p(o)$ to be the probability that the randomized rounding exponential mechanism outputs o, we can therefore write

$$p(o) = \sum_{S \sim [0,1)^{|O|}} \Pr[S] p_S(o)$$

and for any adjacent database, we can write

$$p'(o) = \sum_{S \sim [0,1)^{|O|}} \Pr[S] p'_S(o)$$

where $S \sim [0, 1)^{|O|}$ indicates the set of all possible random values $s \in [0, 1)$ used for sampling. Because $\Delta u_S \leq \alpha$, we have that $\frac{p_S(o)}{p'_S(o)} \leq e^{-2\alpha\varepsilon}$, so

$$\begin{aligned}
\frac{p(o)}{p'(o)} &= \frac{\sum_{S \sim [0,1)^{|O|}} \Pr[S] p_S(o)}{\sum_{S \sim [0,1)^{|O|}} \Pr[S] p'_S(o)} \\
&\leq \frac{\sum_{S \sim [0,1)^{|O|}} \Pr[S] e^{2\alpha\varepsilon} p'_S(o)}{\sum_{S \sim [0,1)^{|O|}} \Pr[S] p'_S(o)} \\
&= e^{2\alpha\varepsilon}
\end{aligned}$$

□

6.2.2 Rust Code

```
/// Randomized Rounding
/// ## Arguments
///   * `x`: the value to round
///   * `arithmetic_config`: the arithmetic configuration to use
/// ## Returns
/// `x` rounded to the nearest smaller or larger integer by drawing a random value
/// `rho` in `[0,1]` and rounding down if `rho > x_fract`, rounding up otherwise.
pub fn randomized_round<R: ThreadRandGen>
                        (x: f64,
                         arithmetic_config: &mut ArithmeticConfig,
                         mut rng: R)
  -> i64
{
    // if x is already integer, return it
    if x.trunc() == x { return x as i64; }

    let x_fract = x.fract(); // fractional part of x
    let x_trunc = x.trunc() as i64; // integer part of x
    // Draw a random value
    let rho = arithmetic_config.get_rand_float(&mut rng);
```

```
    if rho > x_fract {
        return x_trunc; // round down
    } else {
        return x_trunc + 1; // round up
    }
}
```

7 Eta

Source File: params.rs.

7.1 Struct

The Eta struct is very simple, and just holds the values x, y and z needed to compute the parameter $z \log_2(x/2^y)$.

7.1.1 Rust Code

```
/// Privacy parameter of the form `Eta = -z * log_2(x/2^y)` where
/// `x < 2^y` and `x,y,z > 0`.
#[derive(Debug, Copy, Clone)]
pub struct Eta {
    pub x: u32,
    pub y: u32,
    pub z: u32,
}
```

7.2 Constructors

The construction of an Eta privacy parameter is very simple for a specified x, y and z. The check function is used to verify basic properties of the provided x, y and z to ensure that all are positive, and $x/2^y < 1$.

7.2.1 Rust Code

```
    /// Creates `Eta` privacy parameter from the given `x`, `y` and `z`.
    /// ## Returns
    /// `Result<Eta,&str>` with the created `Eta` on sucess or an error string
    /// on failure.
    /// ## Errors
    /// Returns `Err` if `x`, `y`, or `z` do not meet the requirements.
    pub fn new(x: u32, y: u32, z: u32)
        -> Result<Eta>
    {
        let eta = Eta {x, y, z};
        eta.check()?;
        Ok(eta)
    }
```

```
    pub fn check(&self)
        -> Result<()>
    {
        // Check all parameters nonzero
        if self.x == 0 {
            return Err("x must be nonzero".into());
        }
        if self.y == 0 {
            return Err("y must be nonzero".into());
```

```
        }
        if self.z == 0 {
            return Err("z must be nonzero".into());
        }

        // Check x < 2^y
        if self.x > 2u32.pow(self.y) - 1 {
            return Err("x > 2^y - 1".into());
        }
        Ok(())
    }
```

7.3 Base Computation

For convenience, the `Eta` struct can compute $2^{-\eta}$ at a give precision and returns a `Float` representing the value. Note that no inexact arithmetic enforcement is performed. This is intentional, and the caller is responsible for enforcing exact arithmetic if needed. (See, e.g., `get_empirical_precision`.)

```
    /// Get the base `2^(-eta)`
    /// ## Arguments
    ///   * precision: the precision with which to construct the base
    /// ## Returns
    /// Returns an `mpfr::Float` with the requested precision equivalent to
    /// `2^(-eta.z * log_2(eta.x /2^(eta.y)))` or an error.
    /// ## Errors
    /// Returns an error if the `check()` method fails, i.e. not properly initialized.
    pub fn get_base(&self, precision: u32)
        -> Result<Float>
    {
        self.check()?;
        let v = Float::i_exp(self.x as i32, - (self.y as i32));
        let x_2_pow_neg_y = Float::with_val(precision, v);
        let z = Float::with_val(precision, self.z);
        let base = x_2_pow_neg_y.pow(z);
        return Ok(base);
    }
```

8 Additional Proofs

Proposition 8.1. *Given a utility function u such that $\Delta u \leq \alpha$, $\mathsf{clamp}(u, A, B)$ where*

$$\mathsf{clamp}(u, A, B)(x, o) := \min(\max(A, u(x, o)), B)$$

has sensitivity $\Delta\mathsf{clamp}(u, A, B) \leq \Delta u$.

The proof of the proposition follows from observing that clamping values cannot increase the difference in utility of adjacent databases.[4]

9 Miscellaneous

[4]Note that settings in which outcome spaces of the mechanism are not equivalent for adjacent databases, e.g. integer partitions, that this clamping argument does not hold, and the full range of u specified by the mechanism must be supported at the cost of increased precision.

9.1 Canary tests

9.1.1 Arithmetic Flags

The test `test_flags` checks several of the important `mpfr::flags` properties that are critical for correctness of the mechanisms. If this test fails, it should be considered a critical failure, and it's likely that there will be undesired behavior.

A Formatting Information

A.1 Inline Rust Code.

Rust code is included using the `listings` package with a custom Rust style file from `https://github.com/denki/listings-rust`. We can pull directly from the current version of the code file, as long as we have the appropriate line numbers for start/end of the function, or just pull in the whole file. By including the Rust code inline, it should be easier for either one or two reviewers to comment on a single document for pseudocode matching code and proof of pseudocode correctness. In cases where code is extremely simple, this may also remove the need for redundant pseudocode (see: `randomized_round`).