Privacy Proofs for OpenDP: Bounded Sum with Known n

Sílvia Casacuberta, Grace Tian, Connor Wagaman

Version as of September 23, 2021 (UTC)

Contents

1 Versions of definitions documents **1**

2 Current discrepancies between the Rust implementation and the proof **2**

3 Algorithm Implementation **2**
3.1 Code in Rust 2
3.2 Pseudocode in Python 2

4 Proof **4**
4.1 Symmetric Distance 4
4.2 First proof: using the path property (adjacent pairs approach) 6

Warning 1 (Proof only applies to integer types)**.** Please note that this proof only applies for bounded sums calculated on integers. Specifically, the user-specified type `T` (more details on `T` can be found in section 3.2) must be an integer type, such as `u32` or `i64`.
Other types should be used with caution; for example, we know that the current implementation does not work for floats.

1 Versions of definitions documents

When looking for definitions for terms that appear in this document, the following versions of the definitions documents should be used.

- **Pseudocode definitions document:** This proof file uses the version of the pseudocode definitions document available as of September 6, 2021, which can be found at this link (archived here).
- **Proof definitions document:** This file uses the version of the proof definitions document available as of September 6, 2021, which can be found at this link (archived here).

2 Current discrepancies between the Rust implementation and the proof

To guarantee the correctness of the proof, we highlight the discrepancies that this pseudocode and proof have with respect to the actual Rust implementation:

- The pseudocode implementation makes the necessary checks with `checked_mul` to avoid overflow issues when computing the stability relation.
- The Rust code does not impose the restriction for `T` to be an integer type. As noted in the initial warning, the current stability relation only guarantees correctness for integer types.
- The Rust code uses the stability relation $\texttt{d_out} \geq \texttt{d_in} \cdot (\texttt{U} - \texttt{L})/2$ instead of the relation found in the pseudocode, namely $2 \cdot \texttt{d_out} \geq \texttt{inf_cast}(\texttt{d_in}, T) \cdot (\texttt{U} - \texttt{L})$. There is already a PR to resolve this discrepancy: https://github.com/opendp/opendp/pull/315.
- The Rust code does not check for possible overflow errors when performing the arithmetic operation $\texttt{U} - \texttt{L}$, whereas the current pseudocode includes a `checked_sub` for it.
- The Rust code does not include an `inf_cast` in the stability relation, which is necessary for type correctness, but there is already a PR that resolves this discrepancy: https://github.com/opendp/opendp/blob/493fe87273adca3294551faa927039b755c1756e/rust/opendp/src/trans/sum/mod.rs#L65-L94.
- The previous PR also includes the new user-specified type trait `InfDiv`, which the current pseudocode does not include.

These changes will soon be applied to the Rust code. The pseudocode also uses the variable `n` instead of Rust's `size`, which is more conventional for proof writing.

3 Algorithm Implementation

3.1 Code in Rust

The current OpenDP library contains the transformation `make_bounded_sum_n` implementing the bounded sum function with known n. This is defined in lines 29-49 of the file `mod.rs` in the Git repository[1] (https://github.com/opendp/opendp/blob/8bbb0fab1da9b86c50235a36d7026189be43b1ab/rust/opendp/src/trans/sum/mod.rs#L29-L49).

3.2 Pseudocode in Python

We present a simplified Python-like pseudocode of the Rust implementation below. The necessary definitions for the pseudocode can be found in Section 1.

[1] Updated on September 20, 2021.

```rust
pub fn make_sized_bounded_sum<T>(
    size: usize, bounds: (T, T)
) -> Fallible<Transformation<SizedDomain<VectorDomain<BoundedDomain<T>>>, AllDomain<T>, SymmetricDistance, AbsoluteDistance<T>>>
    where T: DistanceConstant<IntDistance> + Sub<Output=T>, for <'a> T: Sum<&'a T> + ExactIntCast<usize> + CheckedMul + CheckNull,
          IntDistance: InfCast<T> {
    let size_ = T::exact_int_cast(size)?;
    let (lower, upper) = bounds.clone();
    if lower.checked_mul(&size_).is_none()
        || upper.checked_mul(&size_).is_none() {
        return fallible!(MakeTransformation, "Detected potential for overflow when computing function.")
    }
    Ok(Transformation::new(
        SizedDomain::new(VectorDomain::new(
            BoundedDomain::new_closed(bounds)?)), size),
        AllDomain::new(),
        Function::new(|arg: &Vec<T>| arg.iter().sum()),
        SymmetricDistance::default(),
        AbsoluteDistance::default(),
        // d_out >= d_in * (M - m) / 2
        StabilityRelation::new_from_constant((upper - lower) / T::exact_int_cast(2)?)))
}
```

Preconditions

To ensure the correctness of the output, we require the following preconditions:

- **User-specified types:**
  - Variable `n` must be of type `usize`.
  - Type `T` must have traits `DistanceConstant(IntDistance)`, `TotalOrd`, `CheckedMul`, `CheckNull`, `Sum(Output=T)`, `Sub(Output=T)`, and `ExactIntCast(usize)`.
  - `IntDistance` must have trait `InfCast(T)`. (Note that this bullet point is not needed in this proof, but it is needed in the code so a hint can be constructed; otherwise a binary search would be needed to construct the hint.)
  - Variables `U` and `L` must be of type `T`, and we must have `L` $\leq$ `U`.

Postconditions

- Either a valid `Transformation` is returned or an error is returned.

```python
def MakeBoundedSumN(L: T, U: T, n: usize):
    input_domain = SizedDomain(VectorDomain(IntervalDomain(L, U)), n)
    output_domain = AllDomain(T)
    input_metric = SymmetricDistance()
    output_metric = AbsoluteDistance(T)

    n_ = exact_int_cast(n, T)
    if checked_mul(L, n_).is_none or checked_mul(U, n_).is_none:
        raise Exception('Potential overflow')

    def relation(d_in: u32, d_out: T) -> bool:
        d_in_ = inf_cast(d_in, T)
        if checked_mul(2, d_out).is_none or checked_mul(d_in_, U-L).is_none
            or checked_sub(U, L).is_none:
            raise Exception('Overflow occurs in the stability relation')
        return 2*d_out >= d_in_ * (U - L)
```

```
17
18     def function(data: Vec[T]) -> T:
19         return sum(data)
20
21     return Transformation(input_domain, output_domain, function,
       input_metric, output_metric, stability_relation = relation)
```

4 Proof

4.1 Symmetric Distance

Theorem 4.1. *For every setting of the input parameters (`L, U, n`) to `MakeBoundedSumN` such that the given preconditions hold, `MakeBoundedSumN` raises an exception (at compile time or run time) or returns a valid transformation with the following properties:*

1. (Appropriate output domain). *For every element v in `input_domain`, `function`(v) is in `output_domain`.*

2. (Domain-metric compatibility). *The domain `input_domain` matches one of the possible domains listed in the definition of `input_metric`, and likewise `output_domain` matches one of the possible domains listed in the definition of `output_metric`.*

3. (Stability guarantee). *For every pair of elements `v`, `w` in `input_domain` and for every pair (`d_in`, `d_out`), where `d_in` has the associated type for `input_metric` and `d_out` has the associated type for `output_metric`, if `v`, `w` are `d_in`-close under `input_metric` and `relation`(`d_in`, `d_out`) = `True`, then `function(v)`, `function(w)` are `d_out`-close under `output_metric`.*

Proof. **(Part 1 – appropriate output domain).** In the case of `MakeBoundedSumN`, this is equivalent to showing that for every vector `v` in `SizedDomain(VectorDomain(IntervalDomain(L, U)), n)`, where `L` and `U` have type `T`, `function(v)` belongs to `AllDomain(T)`.

The output correctness follows from the overflow check done through the `checked_mul` function on line 9 and from the type signature of `function` as defined on line 18.

The check for overflow ensures that `function(v)` is contained within the interval `[get_min_value(T), get_max_value(T)]`, and hence prevents any overflow from occurring in calculating the return value on line 19. The check for overflow works in the following way: first, the size of the vector on which we are taking the bounded sum is `exact_int_cast`ed to a value `n_` of type `T` so that we can work with it (if `n` it cannot be casted exactly – for example, if `n` falls outside the range of values representable by type `T` – a cast error will be returned; we know that the `exact_int_cast` will work because `T` has trait `ExactIntCast(usize)`). Then, we do a `checked_mul` that multiplies `n_` and `L`, and which returns `None` if the result overflows. Because `checked_mul(L,n_)` is the minimum possible sum we could get, and because `checked_mul(U,n_)` is the maximum possible sum we could get, we know that if neither of these cases will result in overflow, then no overflow will occur at any step in our calculation of the final answer. If `None` is returned by the `checked_mul`, indicating that overflow can occur, then line 9 will raise an exception for potential overflow.

The type signature on `function(v)` automatically enforces that `function(v)` has return type `T` and hence is in `AllDomain(T)`, which is set on line 3 as the `output_domain`. Since the Rust code successfully compiles, by the type signature the appropriate output

domain property must hold. Otherwise, the code will raise an exception for incorrect input type.

(Part 2 – domain-metric compatibility). For `MakeBoundedSumN`, proving this part corresponds to showing that `SizedDomain(VectorDomain(IntervalDomain (L, U)), n)` is compatible with `SymmetricDistance()` (see line 4), and that `AllDomain(T)` is compatible with `AbsoluteDistance(T)` (see line 5). The former follows directly from the list of compatible domains in the definition of symmetric distance, as described in the pseudocode definitions document in Section 1. The latter follows directly from the list of compatible domains in the definition of absolute distance, as described in the pseudocode definitions document in Section 1.

(Part 3 – stability guarantee). We begin with a lemma.

Lemma 4.2 (Pseudocode guarantee to arithmetic guarantee)**.** *If `2*d_out >= inf_cast(d_in, T)*(U-L)`, then, when performing true arithmetic, `d_out` $\geq$ `d_in` $*\,(U-L)/2$.*

Proof. Assume that we have `2*d_out >= inf_cast(d_in, T)*(U-L)`. Note that our checked arithmetic operations in lines 13 and 14 have the effect of enforcing that `(U-L)` $= (U-L)$. The operation `inf_cast(d_in,T)` has the property that `inf_cast(d_in,T)` $\geq$ `d_in`. The checked arithmetic also ensures that `inf_cast(d_in,T)*(U-L)` is exact, so we have the guarantee that `inf_cast(d_in,T)*(U-L)` $\geq$ `d_in` $*\,(U-L)$. The checked arithmetic also ensures that `2*d_out` does not overflow, which has the effect of ensuring that `2*d_out` $=$ $2 *$ `d_out`.

Therefore, we have `2*d_out` $= 2 *$ `d_out` $\geq$ `inf_cast(d_in,T)*(U-L)` $\geq$ `d_in` $*\,(U-L)$, so we have $2 *$ `d_out` $\geq$ `d_in` $*\,(U-L)$. We now divide both sides by 2; since we are working with true arithmetic now, the inequality `d_out` $\geq$ `d_in` $*\,(U-L)/2$ is clearly true under our assumption that `2*d_out >= inf_cast(d_in, T)*(U-L)`. This completes our proof that, if `2*d_out >= inf_cast(d_in, T)*(U-L)`, then `d_out` $\geq$ `d_in` $*\,(U-L)/2$. □

Throughout the stability guarantee proof, we can assume that `function(v)` and `function(w)` are in the correct output domain, by the *appropriate output domain property* shown above.

Since by assumption `relation(d_in, d_out) = True`, by the `MakeBoundedSumN` stability relation (as defined in line 11 in the pseudocode), we have that `2 * d_out >= inf_cast(d_in, T) * (U-L)`. Therefore, by lemma 4.2, we also have that `d_out` $\geq$ `d_in` $*$ $(U-L)/2$.

We now show that, if `d_out` $\geq$ `d_in` $*\,(U-L)/2$, then `function(v)`, `function(w)` are `d_out`-close under `output_metric`.

Moreover, note in the statement of theorem 4.1 that `v`, `w` are assumed to be `d_in`-close. By the definition of the symmetric distance metric in the proof definitions document linked in section 1, this is equivalent to stating that $d_{Sym}(v, w) = |\text{MultiSet}(v)\Delta\text{MultiSet}(w)| \leq$ `d_in`. Further, applying the histogram notation,[2] it follows that

$$d_{Sym}(v, w) = \|h_v - h_w\|_1 = \sum_z |h_v(z) - h_w(z)| \leq \texttt{d_in}.$$

We want to show that

$$d_{Abs}(\texttt{function}(v), \texttt{function}(w)) \leq d_{Sym}(v, w) \cdot \frac{\texttt{U-L}}{2}.$$

[2]Note that there is a bijection between multisets and histograms, which is why the proof can be carried out with either notion. For further details, please consult the proof definitions document in section 1.

This would imply that

$$d_{Abs}(\texttt{function}(v), \texttt{function}(w)) \leq d_{Sym}(v, w) \cdot \frac{\texttt{U-L}}{2} \leq \texttt{d_in} \cdot \frac{\texttt{U-L}}{2}, \tag{1}$$

and by the stability relation this will imply that

$$d_{Abs}(\texttt{function}(v), \texttt{function}(w)) \leq \texttt{d_out}, \tag{2}$$

as we want to see. □

4.2 First proof: using the path property (adjacent pairs approach)

To show that $d_{Abs}(\texttt{function}(v), \texttt{function}(w)) \leq d_{Sym}(v, w) \cdot \frac{\texttt{U-L}}{2}$, we will use the three lemmas described in the section "The path property of symmetric distance on sized domains" from Section 5.2 in the document "List of definitions used in the proofs". By lemma 5.7 in the proof definitions document (linked in section 1), which is applicable to `MakeBoundedSumN` because `input_domain` is a a sized domain and `input_metric` is symmetric distance, it suffices to show the following: For all vectors $x, y \in \texttt{input_domain}$ such that $d_{Sym}(x, y) = 2$, it follows that

$$d_{Abs}(\texttt{function}(x), \texttt{function}(y)) \leq \texttt{U} - \texttt{L}.$$

By Lemma 5.6 from "List of definitions used in the proofs", we know that vectors x, y only differ on one element, given that, by assumption, $d_{Sym}(x, y) = 2$. Wlog, let this different element be the k-th element of x and y, where $x_k = \alpha$, $y_k = \beta$ with $\alpha \neq \beta$.[3] Then,

$$d_{Abs}(\texttt{function}(x), \texttt{function}(y)) = |\texttt{function}(x) - \texttt{function}(y)| =$$

$$= \Big| \sum_{i=0}^{\texttt{n}-1} x_i - \sum_{i=0}^{\texttt{n}-1} y_i \Big| = \Big| \sum_{i=0}^{\texttt{n}-1} (x_i - y_i) \Big| = |0 + (x_k - y_k)| = |\alpha - \beta| \leq |\texttt{U-L}| = \texttt{U-L},$$

since $\texttt{U} \geq \texttt{L}$.

[3] The first element of a vector is indexed by 0.