Natural Difference Geometry

by Robert M, Sven Nilsen, 2025

In this paper we introduce a geometric difference calculus on natural numbers.

Natural Difference Geometry is defined as a geometric calculus on functions of the type:

nat → nat

For some function of type `f : nat → nat`, one extracts sequences of natural numbers by starting at some arbitrary initial position `a : nat` and applies `f` to it repeatedly `n` number of times $`f^n(a)`$.

The idea of starting at some arbitrary position is very important. One can think about such functions as a directed graph where a node can have any number of incoming arrows, but every node has a unique outgoing arrow. Since one can start at any location in the graph, it can be thought of as random accessed memory that is readonly. The address of each node is given by some number. So, the number that you look up the node with, is the input to the function. The output number gives you the address of a new node. Using this method, there is no efficient way to get all incoming arrows to some node, but it is efficient to traverse sequentially through output arrows.

A set of standard functions are defined to allow sharing of constructions in this language. When there is a topological transform from any function to some natural construction using functions from this set, one prefers the natural construction as a representation of the topological group. This means, when talking about the topology of this geometry, one is ignoring the numeric values.

In this calculus, there are 3 standard basic functions:

id := \(x : nat) = x	terminal
inc := \(x : nat) = x + 1	open
cyc := \(n : nat) = \(x : nat) = x – (x % n) + (x + 1) % n	closed

The `id` function is considered terminal. It means when one wants to stop some process or create a filter of some kind, one can return the input as output. The other two functions `inc` and `cyc` are used to create open and closed sequences respectively. For example, one can detect whether a sequence is open by adding a constraint that any closed sequence has an upper limit on the size of cycles. This makes it possible for an algorithm to halt after a certain number of steps. Without this constraint, the algorithm can only detect closed sequences and might not halt after finite steps.

There is 1 standard super function:

sup := \(n : nat, f : nat → nat) = \(x : nat) = f(x / n) * n + (x % n) orthogonal

The super function is used to generate orthogonal or parallel sequences to a second function.

	id	**inc**	**cyc(n)**
super(n, id)	*invariant*	*invariant*	*invariant*
super(n, inc)	*invariant*	grid	tower
super(n, cyc(m))	*invariant*	tangled mess	torus

There is 1 standard composition function:

```
comp := \(g : nat → nat, f : nat → nat) = \(x : nat) = g(f(x))
```

It is common to write this as e.g. `g · f` or `g . f` (keyboard friendly notation).

The composition function is commonly used in other parts of mathematics, so it should be familiar to some people. Notice that it the left function gets applied after the right function.

There is a standard even/odd independent pairing function:

```
even_n2 := \(f : nat → nat, g : nat → nat) = \(x : nat) =
    if x % 2 == 0 {2 * f(x / 2)} else {2 * g((x - 1) / 2) + 1}
```

Together with the maps back to individual components:

```
even_n2_fst := \(f : nat → nat) = \(x : nat) = f(2 * x) / 2
even_n2_snd := \(f : nat → nat) = \(x : nat) = (f(2 * x + 1) – 1) / 2
```

Notice that the outputs do not depend on the input of the other, so this is a parallel tuple:

```
f × g
```

To create an n-tuple, one usually nests the right argument, e.g. `f × (g × h)`.

For tuples that requires depending on two arguments, one uses a diagonal space-filling curve:

```
n2 := \(f_x : nat → nat, f_y : nat → nat) = \(i : nat) = {
    let [x, y] = n1_to_n2(i);
    n2_to_n1([f_x(x), f_y(y)])
}
n2_to_n1 := \([x, y]: nat^2) = (x + y) * (x + y + 1) / 2 + y
n1_to_n2 := \(n : nat) = {
    let s = (isqrt(8 * n + 1) – 1) / 2;
    let y = n – (s * (s + 1)) / 2;
    [s – y, y]
}
```

Look for an integer square root function `isqrt` in your programming language's standard library, if it exists, for better numerical accuracy. Otherwise, you might use floating point square root.

This tuple is written with parantheses, e.g. `(f, g)`. N-tuples usually nest right e.g. `(f, (g, h))`.

There exists no map back to individual components due to curvature, but we use uniforms instead:

```
n2_uniform := \(f : nat → nat, uni_y : nat) = \(i : nat) = \(x : nat) =
    n1_to_n2(f(n2_to_n1(if i == 0 {[x, uni_y]} else {[uni_y, x]})))[i]
n2_fst_uniform := \(f : nat → nat, uni_y : nat) = n2_uniform(f, uni_y)(0)
n2_snd_uniform := \(f : nat → nat, uni_y : nat) = n2_uniform(f, uni_y)(1)
```

For example, `h <=> (f, g)`, `n2_fst_uniform(h, 0) <=> f`. You have to specify some uniform parameter because there is no unique way to restore the individual components.

Now, before continuing, it is a good idea to make an overview of three kinds of components:

Type	Notation	Description
$(nat \to nat)^2$	(f, g)	tuple
$(nat^2 \to nat)^2$	f ~> g	embed
$(nat^2 \to nat^2)^2$	f_x, f_y	embedded

It will be a bit to digest here, because there is not just one kind of component, but multiple ones which you use depending on context. When we talk about a tuple `(f, g)`, this means independent components, just like with `f × g`. The encoding is different, but they mean basically the same thing. The difference is that with `f × g` you do not have to worry about more than one kind of component. With `(f, g)`, you get three different kinds and have to keep track of them.

There is 1 standard function for embedding (using parentheses since it is only encoded):

n2_embed := \(f : $nat^{(2)} \to$ nat, g : $nat^{(2)} \to$ nat) = \(i : nat) = n2_to_n1([f(i), g(i)])

When we embed two functions `f ~> g`, both can depend on x and y.
The `f` function decides the output x and the `g` function decides the output y.
Since both can depend on x and y, we can express curvature.
Curvature means that it is not possible to isolate the components into uniform functions.

When we read out spatial differences, we want to use them as embedded inside the space:

```
n2_embedded := \(f : nat → nat) = \(i : nat) = \(p : nat) = {
        let y = n1_to_n2(p)[1 – i];
        let x = n1_to_n2(f(p))[i];
        n2_to_n1(if i == 0 {[x, y]} else {[y, x]})
}
n2_fst_embedded := \(f : nat → nat) = n2_embedded(f)(0)
n2_snd_embedded := \(f : nat → nat) = n2_embedded(f)(1)
```

Although either x or y might be ignored because it is the same as input,
we want to keep both because it makes it easier to reason about spatial differences.

The reason that `embed` uses one output is because there is no easy and well defined method to join the result if they are not perfectly spatial. This means, when we write `(f, g)`, there is no curvature, but when we write `f ~> g` there might be curvature, but it is well defined.

Now, if we write `$f_0$` we mean the spatial difference along the x axis and when we write `$f_1$` we mean the spatial difference along the y axis. However, it would be technically incorrect to write e.g. `$f_0$ ~> $f_1$`, because embedded functions are not the same as the functions we are embedding.

In informal theorem proving, we might ignore the difference between embedding and embedded functions. This makes it a lot easier to write proofs. However, you should be aware of the technical issues and try to avoid ambiguity when it might cause problems for the reader.

To both avoid ambiguity and to solve the ergonomic problems with theorem proving, we might make the language we use more abstract. Here, we can use Homotopy Theory. In Homotopy Theory, a homotopy `h` between `f, g` has the property `h(0) <=> f` and `h(1) <=> g`. Now, from this definition you can immediately see it is biased toward spatial differences, which are embedded.

With other words, in Homotopy Theory we talk about homotopies as some kind of tuples, except instead of just talking about `n` components, we have `n = $2^b$` components for `b` number of bits. This might get confusing sometimes, because we use two different languages to talk about the same number of dimensions. Now, if you are used to logarithmic transforms, then this duality might not seem so bad. It is simply two mathematical perspectives of the same thing. However, the most confusing part can be that in Homotopy Theory, we talk about n-cubes, that here actually means b-cubes, for the number of bits. A line is 1 bit, or a 1-cube. A square is 2 bits, or 2-cube. A cube is 3 bits, or 3-cube. A 4-cube is 4 bits. A 5-cube is 5 bits etc.

The advantage of using Homotopy Theory, is that as the number of dimensions by `n` grows exponentially with the number of bits `b`, we do not have to visualize that many dimensions.
For example, a cube is 8D. It is not easy to visualize 8D, but thinking about cubes is easier.

So, how do we write proofs from the perspective of Homotopy Theory? The secret is that we just say we are using Homotopy Theory and then write the way that we find most convenient. If you want to use tuples, e.g. `($f_0$, $f_1$)`, just go ahead! These tuples have nothing to do with the tuples we use formally. Do not worry about it. You only think about the 3 different kinds when you need to program this using the standard functions. This makes it more convenient for readers, because in one section of the paper, you can use Homotopy Theory and in another section you can use the standard functions to explain how it works in detail.

For example, in space-time we often use 4D. This requires 2 bits, hence a square in Homotopy Theory. One can use `x := 00, y := 01, z := 10, t := 11`. So, `$f_x$` becomes `$f_{00}$`, which translates to:

```
n2_fst_embedded(n2_fst_embedded(f))
```

Here are all 4 dimensions:

f_x	f_{00}	n2_fst_embedded(n2_fst_embedded(f))
f_y	f_{01}	n2_snd_embedded(n2_fst_embedded(f))
f_z	f_{10}	n2_fst_embedded(n2_snd_embedded(f))
f_t	f_{11}	n2_snd_embedded(n2_snd_embedded(f))

Now, if you have `f <=> ((inc, inc), (inc, cyc(10)))`, then time loops in steps of 10.
It is not technically correct to say that `$f_t$ <=> cyc(10)`, because that cycle is embedded:

```
f_t <=> \(x : nat) = {
        let p2 = n1_to_n2(p);
        let p3 = n1_to_n2(p2[1]);
        n2_to_n1([p2[0], n2_to_n1([p3[0], cyc(10)(p3[1])])])
}
```

Notice that `cyc(10)` is buried deep inside `$f_t$`.

When we do Homotopy Theory, we might just write `$f_t$` and not care about the formal issues. Just try to make proofs easy to read. If we want to swap two dimensions, we can use `transpose_map`:

```
transpose := \(i : nat) = {
        let [x, y] = n1_to_n2(i);
        n2_to_n1([y, x])
}
transpose_map := \(f : nat → nat) = transpose . f . transpose
```