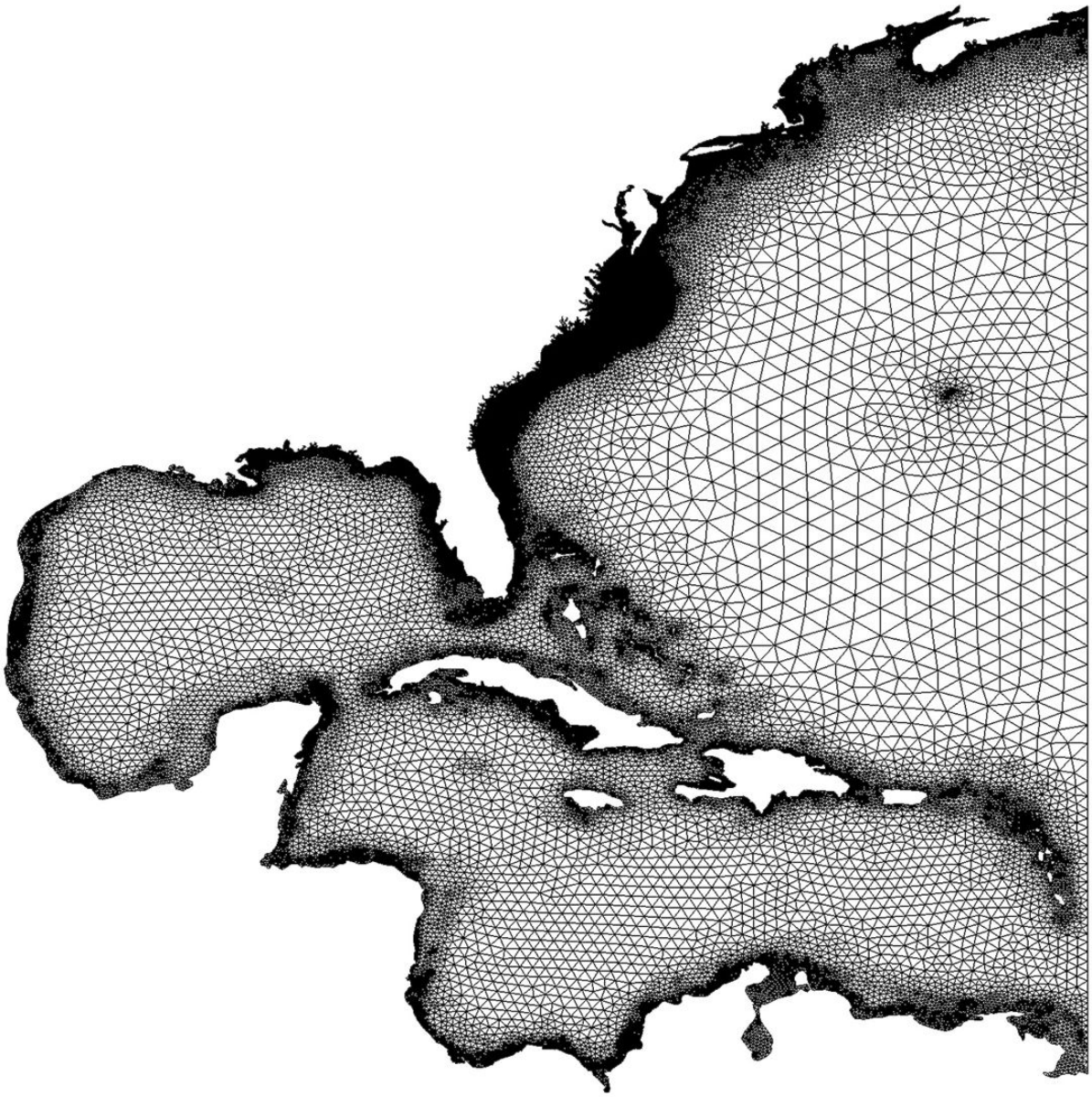
Forge Lab 4: Logic For Systems and Scientific Computing

This is the lab that corresponds to Tristan Dyer's lecture last week on the role of lightweight formal methods in scientific computing. The lecture slides can be found here.

Statics: Representing a Mesh

Finite element methods work by discretizing a continuous domain and approximating a solution over it with piecewise polynomial functions. This discretization results in a mesh of elements and nodes, and can be thought of as a triangulation of a surface. The following image shows a finite element mesh that is used to simulate hurricane storm surge. Note that only oceans/seas and coastlines are represented by the mesh, as that is where the water is!



As the image shows, the land and seafloor surfaces are represented as a collection of contiguous, non-overlapping triangles, or *elements*, that meet along their edges and at their vertices, or *nodes*. This particular mesh has approximately 620,000 nodes and over 1.2 million elements.

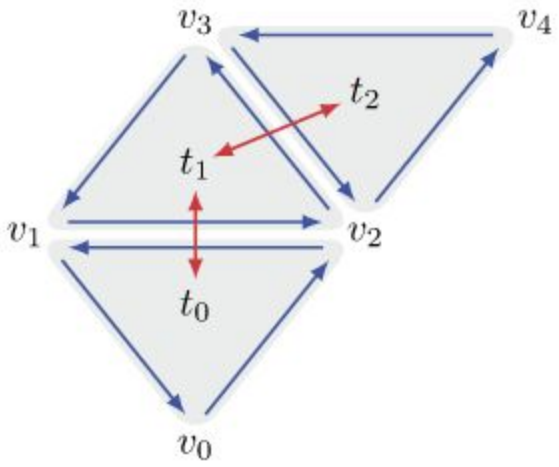
In the first part of our model we build a representation of mesh topology — with triangles and vertices as basic components — that we will later extend with additional attributes as dictated by the algorithm you'll be modeling.

```
sig Mesh {
  triangles: set Triangle,
  adj: set Triangle -> Triangle
}

sig Triangle {
  edges: set Vertex -> Vertex
}


sig Vertex {}
```

The model also includes quite a few predicates that we use to define valid topologies for a triangulation, but it is most important that you understand these three abstractions.

In the declarations above, the signature **Mesh** has a field **triangles** that defines a binary relation over **Mesh** and **Triangle**; this defines the complete set of triangles that constitute the mesh. An additional field **adj** defines a ternary relation that contains the tuple *(m, s, t)* when a mesh *m* includes "adjacent" triangles *s* and *t*. Similarly, in signature **Triangle**, the field **edges** is a relation that contains the tuple *(t, u, v)* when a triangle *t* includes a directed edge from vertex *u* to *v*. The **Vertex** signature, on the other hand, contains no fields and therefore defines no additional relations.

By way of example, the following instance shows a mesh with three triangles and five vertices that is consistent with the declarations above.



Dynamics: Wetting and Drying

In a hurricane storm surge simulation, the wetting and drying algorithm is used to determine which nodes and elements are wet, and therefore participating in the simulation, and which are dry, and therefore *not* participating in the simulation. To better understand what a wetting and drying algorithm is, and why we need to use one, briefly explore this Observable notebook.

The wetting and drying algorithm you will be modeling in this lab comes from a storm surge simulation software package called ADCIRC. In ADCIRC, a *node* can be viewed as a vertex located in three-dimensional space with x, y, z coordinates along with other properties such as water surface elevation, water velocity, and wet-dry state. An *element* can be viewed as a triangle in a plane defined by its incident nodes (the three nodes that define the triangle), along with its own wet-dry state and physical quantities that vary in time.

In terms of implementation, ADCIRC's wetting and drying algorithm produces a series of state changes in nodes and elements at each time step before final wet-dry states can be determined. There are a number of topological dependencies that, combined with those state changes, complicate reasoning about the wet-dry algorithm. For example, element wet-dry state depends on the wet-dry state of its incident nodes, while nodal wet-dry state depends on water surface elevation, incident elements, and adjacent nodes.

The ADCIRC wetting and drying algorithm consists of 6 discrete steps, defined below, and relies on the following nodal and elemental properties, along with a few user-defined constants:

- Properties of node *n*:
  - $\mathbf{W}_n$ — the wet/dry status of node ***n***
  - $\mathbf{W}^t_n$— the temporary wet/dry status of node ***n***
  - $\mathbf{H}_n$ — the water surface elevation at node ***n***
- Elemental properties:
  - $\mathbf{wet}_e$ — the wet/dry status of element ***e***
  - $\mathbf{V_{ss}(e)}$ — the steady-state water velocity in element ***e***
- Constants:
  - $\mathbf{V}_{min}$ — a constant defining “slow flow” across an element
  - $\mathbf{H}_{min}$ — a constant defining the minimum water column required to make nodes and elements dry

0: **for** e **in** *elements* **do** ▷ initialization: start with all elements being wet
  $wet_e \leftarrow$ true
1: **for** n **in** *nodes* **do** ▷ make nodes with low water column height dry
  if W_n **and** $H_n < H_{min}$ **then**
    $W_n \leftarrow$ false, $W^t_n \leftarrow$ false
2: **for** e **in** *elements* **do** ▷ propagate wetting across element
  if $\neg W_i$ for exactly one node i on e **and** $V_{ss}(e) > V_{min}$ **then** unless flow is slow
    $W^t_i \leftarrow$ true
3: **for** e **in** *elements* **do** ▷ allow water to build up
  find nodes i and j of e with highest water surface elevations η_i and η_j on an incline
  if $\min(H_i, H_j) < 1.2H_{min}$ **then**
    $wet_e \leftarrow$ false
4: **for** n **in** *nodes* **do** ▷ make landlocked nodes dry
  if W^t_n **and** n on only inactive elements **then**
    $W^t_n \leftarrow$ false
5: **for** n **in** *nodes* **do** ▷ set the final wet-dry state for nodes
  $W_n \leftarrow W^t_n$

Before modeling individual parts of the algorithm, we first extend the `Triangles` and `Vertex` signatures to include “static” properties.

```
abstract sig Bool {}
```

```
one sig True extends Bool {}
one sig False extends Bool {}

abstract sig Height {}
one sig Low extends Height {}
one sig Med extends Height {}
one sig High extends Height {}

sig Node extends Vertex {
  H: one Height
}

sig Element extends Triangle {
  slowFlow: one Bool,
  lowNode: one Node

}
```

To represent the water column height relative to the value of $\mathbf{H_{min}}$, we introduce an abstract `Height` signature and three extensions `Low`, `Medium`, and `High`. These represent the three water column heights relevant to the algorithm: low ($H_n < H_{min}$), medium ($H_{min} \leq H_n < 1.2H_{min}$), and high ($H_{min} \leq H_n$).

For an element, *e*, a flow of water *e.slowFlow* is true when $\mathbf{V_{ss}(e) \leq V_{min}}$, as used in part 2 of the algorithm. The value *e.lowNode* is the element's node with the *lowest* water surface elevation — or one of the lowest in the case it is not unique — supporting the test in part 3 of the algorithm.

Finally, we introduce a `State` signature that includes all of the variables that go through a state change in an execution of the algorithm.

```
sig State {
    W: set Node -> Bool,
    Wt: set Node -> Bool,
    wet: set Element -> Bool

}
```

Lab Exercises

Before you begin, there's a new version of Sterling that has some really nice features that will make visualizing this lab a lot easier. We recommend that you update forge to the most recent version - some details about how to use these features can be found here: https://piazza.com/class/k47gct456or1u6?cid=484

The following two files contain the same incomplete model of the wetting and drying algorithm. One file contains extensive commentary that should help provide additional context if needed, while the other only provides minimal comments.

- `wetdry-commentary.rkt` - With commentary
- `wetdry.rkt` - Without commentary

Starting with the incomplete model, your task is to complete it by filling in the predicates that represent the individual steps of the wetting and drying algorithm (see section `The steps of the wet/dry algorithm`). The predicate `part0` is provided to establish the initial conditions, and you must complete predicates `part1` through `part5`. You'll note that in each step you are provided with the necessary setup conditions.

Make sure to check out the `Helper predicates` section (right above the preds you are filling in). The `make_wet` and `make_dry` functions will prove especially useful when defining your `implies` statements in parts 2 and 4. Note also the `dom` function (provided near the top of the file), which returns the domain of a binary relation.

Once you have completed the model, run the commands `allWetToDry` and `allDryToWet` to answer the following two questions (leave a "yes" or "no" in a comment above the respective commands in the model):

- Can a mesh start out with all wet nodes and have them all become dry?
- Can a mesh start out with all dry nodes and have them all become wet?

Finally, the command `answer` can be used to answer a question asked on the ADCIRC mailing list: *why would an element with three wet nodes be dry?*

Run the command and explore the instance generated. It is helpful to project over `answer`, `Mesh`, and `State` to get a better view of the complete state of the mesh at each point in the

algorithm. Which step of the algorithm causes an element with three wet nodes to become dry? Leave your answer in a comment above the `answer` command.