Snapview Challenge

This is the implementation of solution for the problem given in the recruitment process for the position of Rust programmer (August 2020). Program is not fully polished, several improvements are possible - see respective section.

Problem

Write a program in *Rust* that calculates the water level in different parts of a landscape. The landscape is defined as positive numbers.

```
10|
9 |           *
8 |         * *
7 |         * *
6 |     *   * *
5 |     *   * *
4 |     * * * *
3 | *   * * * *
2 | *   * * * *
1 | * * * * * *
0 ------------
  0 1 2 3 4 5 6
```

Then it begins to rain. Per hour one unit of rain falls on each segment of the landscape.
Water that falls on segment 3 in the above sample landscape will flow equally into segment 2 and into segment 4, until segment 4 is filled up to the level of segment 3.
Water that falls on segment 5 will flow into segment 4, because segment 6 is higher. Water that falls on segment 6 will flow into segment 5 (and then further into segment 4) because right to segment 6 is an infinite wall (same as left to segment 1).
To the very left and to the very right of the landscape are infinite walls, i.e. water does not flow outside of the defined landscape.

The program shall calculate the water levels for each segment after x number of hours of rain.
The user of the program shall be able to define the landscape when running the program and shall also be able to define the number of hours it will rain.

Describe an algorithm (in text form) and prove that the algorithm works. Then implement the algorithm in the *Rust* programming language.

Solution

1) Data Model.

Landscape is represented by vector of points. Every point has its constant index within landscape. Point reflects small area of landscape treated as unit of calculation (called section in above description). Point object consists of constant ground level which reflects initial state of landscape (shown on graphic) and

variable water amount which is initially zero and then is being increased on rain event and then water flows from one point to other according to described rules. It is assumed that water exchange is loss-less and water does not penetrate into ground. **Water level** (returned by point's `get_height()` function) is understood as ground level plus water amount. It is assumed that area of every point is the same, so when water flies from one point to other respective levels are increased/decreased by the same value. Topology (i.e. from which point to which water can flow) is determined by landscape `neighbors(idx)` function. Current implementation follows task description, but algorithm is flexible and should work in more complicated topologies (e.g. 2D with 4 neighbors or even some arbitrary ones). Water and ground levels are stored as `f64` values.

2) Algorithm.

Program after it read input data and initialize landscape object, calls `rain(...)` function on landscape given number of times presenting water level for each point after every call and then exits.

Every 'rain' function call consists of 3 major steps:

- add water to every point
- simulate water flow (i.e. decreases / increases water amount) between points which are neighbors - function `stabilize_water()`
- optionally fills and returns buffer with water level for each point

Water flow simulation (middle point above) works in following way:

- Repeat below steps until (@) in the iteration there were no water flows (i.e. landscape state did not changed)
    - for every point do following:
        - for every neighbor of the point:
          ~ check if its water level if lower then the point itself; if yes then marks neighbor as **dirty** (adds to `send_water_to` vector)
        - calculates **equal_fraction** as amount of water in the point divided by number of dirty neighbors
        - for every dirty neighbor of the point (@@):
          ~ sets **diff** as difference between current (accounting changes done in previous runs of this loop) 'water level' of point and 'water level' of neighbor
          ~ if 'diff' is less or equal to zero - continue loop with next dirty neighbor (@@@)
          ~ sets **flow_amt** as minimum of half of 'diff' and 'equal_fraction`
          ~ decrease water in the point by 'flow_amt'
          ~ increase water in the neighbor by 'flow_amt'
          ~ optionally record this flow event for debugging purposes
    - optionally call `calc_state()` function and check if state value decreased since previous step (check correctness of algorithm).

Implementation detail:

Calculations are being done on floating numbers. To mitigate rounding effects and ensure calculations stability all water level comparisons are being done with arbitrary tolerance determined by constant `VISCOSITY_COEF` defined in the program as 0.01.

Correctness.

A) Steps described in inner loop (@@) does implement rules provided in the task description. Taking 'half of diff' into account guarantee that water flow will stop when levels equalize. Taking 'equal_fraction' into account causes equal ware distribution in case it can flow in several directions. Setting 'flow_amt' as minimum assures that we do not flow-out more water than it is currently in the point (i.e. we do not reach negative water amount). Also note that this loop is the only one place in the program where water flow occurs.

B) Short circuit of iteration marked above as (@@@) can only happen if there was water flow event in previous loop iteration (which decreased water in the point), otherwise neighbor would not be dirty. This assures that highest level procedure loop (@) will be run again and will cause water flow back from neighbor to the point. We must not implement reverse water flow here (when 'diff' < 0), because this would be break simulation rules (water may flow-out from neighbor also to other points).

C) Major loop marked (@) above will eventually end. This can be shown by introducing state function, for example as:

$$S(t) = \sum_p w_p(t)^k$$

where: sum is for all points in the landscape, w_p is water level of point p, t can be interpreted as counter of major loop, and k is any real number greater than one ($k > 1$).
Note that operations described in inner loop (@@) (the only one place where water levels are changed) decreases value of our state function. Let's proof that for the case $k = 2$.

Let's denote: w_p - water level of point, w_n - water level of neighbor, f - calculated flow amount to transfer. From the code of program it holds:

$$w_n < w_p$$

$$0 < f \leq \frac{w_p - w_n}{2}$$

Respective code:

```
// w_p -> points[pi].water
// w_n -> points[*ni].water
// f -> flow_amt
// VISCOSITY_COEF == 0.01
let equal_fraction = pw / send_water_to.len() as PointHeight;
for ni in &send_water_to {
    let diff = self.points[pi].get_height() -
self.points[*ni].get_height();
    if diff > VISCOSITY_COEF {
        let flow_amt = if equal_fraction < diff / 2.0 { equal_fraction }
else { diff / 2.0 };
        // ...
        self.points[pi].water -= flow_amt;
        self.points[*ni].water += flow_amt;
        idle = false;
```

```
        }
    }
```

Also from task / input data constraints we know that all $w_\bullet$ values are positive or zero.
Assuming we have only one water flow (from p to n) in $t+1$ iteration, our state functions are equal:

$$S(t) = C + w_p^2 + w_n^2$$

$$S(t+1) = C + (w_p - f)^2 + (w_n + f)^2$$

so

$$S(t+1) - S(t) =$$

$$(w_p - f)^2 + (w_n + f)^2 - w_p^2 - w_n^2 =$$

$$w_p^2 - 2w_p f + f^2 + w_n^2 + 2w_n f + f^2 - w_p^2 - w_n^2 =$$

$$2(w_n - w_p)f + 2f^2 \leq$$

$$-(w_p - w_n)^2 + \frac{(w_p - w_n)^2}{2} =$$

$$-\frac{(w_p - w_n)^2}{2} < 0$$

So our state function is strictly decreasing in every major loop iteration. It is also bounded from bottom by zero (and the sum similar to state function but using ground levels). So from well known math calculus theorem state function must converge to some limit - what in practice means that water levels are stable, does not change in next iteration and our loop end. To prevent waiting long time for this stabilization, constant `VISCOSITY_COEF` was introduced. Its value can be decreased by programmer, what would increase precision, but decrease performance. It should not be however set to 0.0.

Computational complexity.

Exact computational complexity is hard to estimate, because it is not obvious how many iterations will be done in highest level loop in water stabilization function. I did not spent too much time on investigation. Maximal possible convexity is $O(N^2)$ and minimal is $O(N)$. Empirical measurements against different random data sets at different sizes shows that it is rather linear $\sim 5N$.

Variants.

There are 2 variants of above algorithm, implemented in modules `simul_manual_1th_v1` and `simul_manual_1th_v2`. The second one is small improvement of first. The only difference is that in second algorithm points in middle loop are processed in order from higher ones to lower ones unlike the first one where they are processed in data driven order. I counted that this change would cause smaller number of highest level iteration. Measurements against large data showed small performance improvement, but not very big, and second algorithm has some memory penalty for additional vector of N size with indexes.

One can switch between two algorithms by emailing respective line in the code near begin of `main.rs` file.

```
fn solver_factory(points_heights: Vec<PointHeight>) -> impl Solver {
    // simul_manual_1th_v1::Landscape::create(points_heights)
    simul_manual_1th_v2::Landscape::create(points_heights)
}
```

3) Program

The main program requires one numeric command line parameter which stands for number of rain simulations to do. Landscape definition is being read from `stdin` as simple stream with one point height (integer or float) in one line. Reading is finished when either end-of-file or empty line is read. When input stream is not redirected program does not print any prompt and user have to just type some values and end with empty line or Ctrl-D.
Program prints results to `stdout` comma separated water levels for points in input data order (one line after each rain simulation). Such data format was easiest to code and convenient for tests - as test program for joy, user friendless was not a priority here.

Usage:

```
Usage:
   sv_challenge N [<input.txt] [>output.txt]
where:
   N          - finish after this number of rain simulations (hours in the
task description)
   input.txt  - text file with landscape definition: one landscape point
with float hight in one line
   output.txt - results - at every simulation step (hour) a line is printed
with comma separated water hights per point in input file order
```

Cargo Features

Each of following two features causes that state function is being calculated at every iteration of highest level loop and it is checked that state value actually decreases.

Feature	Default	Description
`state_fun_f64`	off	Use state function besed on `f64`
`state_fun_bd`	off	Use state function besed on `BigDecimal` (higher precision, but huge performance degradation)

.

Utilities

Inside `examples` folder there is a program that can generate input file for the program with random heights.

```
Usage: sample <points_num> [upper_bound=100]
```

For example, to generate file with 10k points, invoke:

```
cargo run --example sample -- 10000 >data/sample.txt
```

Possible improvements

The time for exercise was limited (few days), so there is still some room for potential improvements in the future. Things that came to my mind are:

- Few thinks noted in the code as `TODO`.
- Unit and integration tests !!!
- It is possible to introduce some parallelism.
    - At first `rayon` crate could be used to add water during rain to points and produce results (steps 1 and 3 of `rain` function).
    - Water stabilization function can be I think parallelized in following way. As in 'v2' algorithm we sort points by ground level and at first process first N highest points. Then if after this step some points became out of water, then process all points between those 2 in separate thread.
    - Other possibility is to divide landscape at some arbitrary points (highest ones) and process them in separate threads. If it will be a need to flow water across threads border implement some locking mechanisms for points 'at border' and process points 'inside area' without locks as they are touched by only one thread.

License

Author

Grzegorz Wierzchowski gwierzchowski@wp.pl