COMP 212 Fall 2022
Homework 08

In this homework you will implement n-body simulation using the Barnes-Hut algorithm.

See the instructions at the end of this document for how to compile and run your code: because the project consists of many files, you will need to use `CM.make` rather than `use`.

1 Sequence Library

For this assignment, you will use the implementation of sequences that you downloaded in lab. It is important that you unzip the homework code so the `src` directory from sequences and `hw08-handout` are next to each other. You can find the signature for sequences in the files `src/sequence/sequencecore-sig.sml` and `src/sequence/sequence-sig.sml`. For your reference, we describe the functions you should use here:

- `Seq.length : 'a Seq.seq -> int`
  `Seq.length s` evaluates to the number of items in `s`.
- `Seq.empty : unit -> 'a Seq.seq`
  `Seq.empty ()` evaluates to the sequence of length zero.
- `Seq.cons : 'a * 'a Seq.seq -> 'a Seq.seq`
  If the length of `xs` is `l`, `Seq.cons (x, xs)` evaluates to a sequence of length `l+1` whose first item is `x` and whose remaining `l` items are exactly the sequence `xs`.
- `Seq.singleton : 'a -> 'a Seq.seq`
  `Seq.singleton x` evaluates to a sequence of length 1 where the only item is `x`.
- `Seq.append : 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`
  If `s1` has length l_1 and `s2` has length l_2, `Seq.append` evaluates to a sequence with length $l_1 + l_2$ whose first l_1 items are the sequence `s1` and whose last l_2 items are the sequence `s2`.
- `Seq.tabulate : (int -> 'a) * int -> 'a Seq.seq`
  `Seq.tabulate (f, n)` evaluates to a sequence `s` with length `n` where the i^{th} item of `s` is the result of evaluating `(f i)`. `Seq.tabulate (f, i)` raises `Range` if `n` is less than zero.

- `Seq.nth : int * 'a Seq.seq -> 'a`
  `nth i s` evaluates to the i^{th} item in `s`. This is zero-indexed. `Seq.nth (i, s)` will raise `Range` if `i` is negative or greater than `(Seq.length s)-1`.

- `Seq.filter : ('a -> bool) * 'a Seq.seq -> 'a Seq.seq`
  `Seq.filter (p, s)` returns the longest subsequence `ss` of `s` such that `p` evaluates to `true` for every item in `ss`.[1]

- `Seq.map : ('a -> 'b) * 'a Seq.seq -> 'b Seq.seq`
  `Seq.map (f, s)` maps `f` over the sequence `s`. That is to say, it evaluates to a sequence `s'` such that `s` and `s'` have the same length and the i^{th} item in `s'` is the result of applying `f` to the i^{th} item of `s`.

- `Seq.reduce : (('a * 'a) -> 'a) * 'a * 'a Seq.seq -> 'a`
  `Seq.reduce c b s` combines all of the items in `s` pairwise with `c` using `b` as the base case. `c` must be associative, with `b` as its identity.

- `Seq.mapreduce : ('a -> 'b) * 'b * ('b * 'b -> 'b) * 'a Seq.seq -> 'b`
  `Seq.mapreduce l e n s` is equivalent to `Seq.reduce n e (Seq.map l s)`.

- `Seq.toString : ('a -> string) * 'a Seq.seq -> string`
  `Seq.toString (ts, s)` evaluates to a string representation of `s` by using `ts` to convert each item in `s` to a `string`.

- `Seq.repeat : int * a -> 'a Seq.seq`
  `Seq.repeat (n, x)` evaluates to a sequence consisting of exactly `n`-many copies of `x`.

- `Seq.flatten : 'a Seq.seq Seq.seq -> 'a Seq.seq`
  `Seq.flatten ss` is equivalent to `reduce append (empty ()) ss`

- `Seq.zip : ('a Seq.seq * 'b Seq.seq) -> ('a * 'b) Seq.seq`
  `Seq.zip (s1,s2)` evaluates to a sequence whose n^{th} item is the pair of the n^{th} item of `s1` and the n^{th} item of `s2`.

- `Seq.split : int * 'a Seq.seq -> 'a Seq.seq * 'a Seq.seq`
  If `s` has at least `i` elements, `Seq.split (i, s)` evaluates to a pair of sequences `(s1,s2)` where `s1` has length `i` and `Seq.append (s1, s2)` is the same as `s`. Otherwise it raises `Range`.

- `Seq.take : int * 'a Seq.seq -> 'a Seq.seq`
  `Seq.take (i, s)` evaluates to the sequence containing exactly the first `i` elements of `s` if $0 \leq i \leq$ `length s`, and raises `Range` otherwise.

- `Seq.drop : int * 'a Seq.seq -> 'a Seq.seq`
  `Seq.drop (i, s)` evaluates to the sequence containing all but the first `i` elements of `s` if $0 \leq i \leq$ `length s`, and raises `Range` otherwise.

[1]Here we use the term "subsequence" to mean any subsequence of a sequence, not necessecarily one whose elements are consecutive in the original sequence. For example, $\langle\rangle$, $\langle 3\rangle$, and $\langle 2,4\rangle$ are subsequences of $\langle 1,2,3,4\rangle$.

2 n-Body Simulations

2.1 Two Planes

2.1.1 Big Picture

The main portion of this programming assignment is modeling movements of bodies through a universe represented by a two-dimensional Euclidian plane. To make this model, we must pick an SML representation of points in the plane that allows us to meaningfully measure the distance between points—that is to say, we must pick a way to measure the universe.

The obvious choice, and the one we made in the lecture on n-body simulation, is to say that a point in the plane is a pair of real numbers, represented by a pair of values of type `real`. Values of type `real` are relatively fast to compute with, come with many helpful functions because they are built into SML, and work well with the visualizer we wrote. Critically, though, they are a floating point precision approximation to real numbers—*not* actual real numbers in the mathematical sense. In particular, addition and multiplication on values of type `real` are not always associative, and multiplication does not always distribute over addition. This means that you can do the same sequence of operations in two slightly different orders and get drastically different results:

```
- 10E30 + (~10E30 + 1.0);
val it = 0.0 : real
- (10E30 + ~10E30) + 1.0;
val it = 1.0 : real
```

This makes testing programs that compute with `real`s hard. Two completely correct implementations of a particular algorithm that use slightly different associations will, in general, produce different results.

A less obvious representation is to say that a point in the plane is a pair of rational numbers. Rationals are not built into SML, but can be represented as pairs of infinite precision integers, where the pair (n, d) is thought of as the fraction $\frac{n}{d}$. These do represent mathematical rational numbers, so all the operations that should be associative and distributive actually are. Critically, this means that the results produced by rational arithmetic are easily testable: any correct implementation of an algorithm using rationals will produce the same output.

There is, however, a price to pay: computation involving values of type rationals is often very slow because the type is implemented with arbitrary precision integer arithmetic. It's so slow that you can't use a simulation implemented using rationals to simulate anything very large or very interesting. More damningly, abstract mathematical rational numbers don't support Euclidian distance: if x and y are rational numbers, it is not necessarily the case that

$$\sqrt{x^2 + y^2}$$

is a rational number. Another definition of distance—*the Manhattan Distance*—is an appropriate definition of distance mathematically speaking in that it makes pairs of rationals into a metric space, but does not intuitively model the universe we know. Manhattan distance between (x_1, y_1) and (x_2, y_2) is $|x_2 - x_1| + |y_2 - y_1|$ — i.e. it describes how far "over" and "up" you have to drive.

2.1.2 Avoiding A Choice

So we have two possible ways to represent points in space, each with benefits and drawbacks. Instead of picking one over the other, we'll use both as they suit us. The reason that this is valid is that both pairs of rationals and pairs of reals form a mathematical object called a metric space: if we only program using properties of metric spaces in general, it doesn't matter which particular metric space we choose when we run the code.

To that end, you will write your code for this assignment without committing to either representation of points and using only functions that work on an abstract idea of points in general.

For testing, we have included implementations of a plane built from both real and rational points. This gives you two choices:

- If you want to run your code to see if your algorithm is correct: use the rational plane and compare your output to ours. (Details in Section 2.5.)

- If you want to run your code, once you've decided that it's correct, to see the results of the simulation, or if you want to see how far off your code is: use the real plane and upload the output into the visualizer. (Details in section 2.5.4.)

Your grade for this assignment will factor in the behaviour of your code on *both* representations of the plane. We will test that your implementation produces exactly the same results as ours with rational points and also that it produces roughly the same results in the real metric space.

2.2 The Plane

To make your code work with both implementations of the plane, we use the SML module system.

2.2.1 Scalars

There is a signature `SCALAR` for numbers (see `scalar.sig`), with two implementations, one in terms of rationals, and the other in terms of reals. The real implementaiton uses the normal Euclidean distance metric, and the rationals implementation uses Manhattan distance. The type `Scalar.scalar` refers to the currently selected implementation. The implementation of real scalars is in `realplaneargs.sml` and of rational scalars is in `realplaneargs.sml`. This provides only some core operations that differ between the two implementations; operations that are the same for both implementations are in `scalar.sml`. Overall, you should be able to do the homework just by reading `scalar.sig` and not the implementations, but look at `realplaneargs.sml` and `scalar.sml` if you are confused about what an operation does (the rationals implementation is a little harder to read). We describe some of the operations in more detail here:

The type `Scalar.scalar` is equipped with the following functions:

- `Scalar.plus : Scalar.scalar * Scalar.scalar -> Scalar.scalar` which computes the sum of two scalars.

- `Scalar.minus : Scalar.scalar * Scalar.scalar -> Scalar.scalar` which computes the difference of two scalars.
- `Scalar.times : Scalar.scalar * Scalar.scalar -> Scalar.scalar` which computes the product of two scalars.
- `Scalar.divide : Scalar.scalar * Scalar.scalar -> Scalar.scalar` which computes the quotient of two scalars.
- `Scalar.compare : Scalar.scalar * Scalar.scalar -> order` which computes the ordering between two scalars.
- `Scalar.fromRatio : IntInf.int * IntInf.int -> Scalar.scalar`
  `Plane.s_fromRatio (x,y)` evaluates to the value of type `Scalar.scalar` which represents $\frac{x}{y}$.
- `Scalar.toString : Scalar.scalar -> string` which computes a string representation of a scalar.

There are other helper functions in the file implemented in terms of these. By using only the above operations, your code will work with either implementation of the plane.

2.2.2 Points and Vectors

Using scalars, we have provided an implementation of the plane, which consists of points and vectors. These are represent by the types `Plane.point` and `Plane.vec`. In order to write our implementation of the Barnes-Hut algorithm, we need several operations on vectors and points in space, many of which we discussed in lecture. The type `Plane.point` is used to represent a point in space, and the type `Plane.vec` is used to represent vectors of velocity, acceleration, etc. In the implementation, we define the type of points and vectors as in lecture:

```
type Plane.point = Scalar.scalar * Scalar.scalar
type Plane.vec = Scalar.scalar * Scalar.scalar
```

This uses `Scalar.scalar` for numbers and represents points and vectors in Cartesian coordinates. In the client, these types are *abstract*, and you should code only in terms of the provided operations on points and vectors.

You can see the full signature for the plane in `space.sig`. Some operations include:

- `Plane.--> : Plane.point * Plane.point -> Plane.vec`
  the vector whose tail is the first point and whose head is the second point
- `Plane.zero : Plane.vec`
  the zero vector
- `Plane.++ : Plane.vec * Plane.vec -> Plane.vec`
  add two vectors

- `Plane.** : Plane.vec * Scalar.scalar -> Plane.vec`
  scale a vector by a constant

- `Plane.origin : Plane.point`
  the origin point of the vector space.

- `Plane.distance : Plane.point * Plane.point -> Scalar.scalar`
  `Plane.distance p1 p2` evaluates to the distance between the points `p1` and `p2`.

- `Plane.midpoint : Plane.point * Plane.point -> Plane.point`
  `Plane.midpoint p1 p2` evaluates to the midpoint of the points `p1` and `p2`.

- `Plane.head : Plane.vec -> Plane.point`
  `Plane.head v` evaluates to the point that corresponds to the displacement of `v` from the origin.

In the Barnes Hut file, we have *opened* the `Plane` module, which means you can refer to these operations without writing `Plane.X`. We have also made the symbols infix, so you can write e.g. `p1 --> p2` and `v1 ** c`.

2.2.3 Bounding boxes

The type `BoundingBox.bbox` represents a rectangular region in two-dimensional space. You will want to use the functions whose types and specs are given in `bbox.sig`. These functions are implemented in `bbox.sml`. In the Barnes Hut file, we have renamed `BoundingBox` to `BB` for conciseness. Here some useful functions:

- `BB.contained :  (bool * bool * bool * bool) * Plane.point * BB.bbox -> bool`
  `BB.contained bs p bb` evaluates to `true` if and only if the point `p` is in the box `b`. The four booleans control whether the left/right/top/bottom edges of the box are included or excluded, where `true` means to exclude an edge.

  For example, `BB.contains((false,false,false,false),p,b)` returns true if `p` is in the box `b` including all of the edges, while `BB.contains((true,false,false,false),p,b)` is the same except it will return false if `p` is directly on the left edge of the rectangle. The order of the booleans is

  (exclude left side, exclude right side, exclude top side, exclude bottom side)

  and a corner is excluded if either of the sides containing it are excluded.

- `BB.diameter :  BB.bbox -> Scalar.scalar`
  Computes the diameter of the box, i.e. the length of the diagonal.

- `BB.from2Points :  Plane.point * Plane.point -> BB.bbox`
  `from2Points (p1, p2)` returns the smallest bounding box containing both `p1` and `p2`

- `BB.fromPoints : Plane.point Seq.seq -> BB.bbox`
  Computes the minimum bounding box containing every point in a sequence of points, assuming the sequence is non-empty.
- `BB.center : BB.bbox -> Plane.point`
  Computes the center point of the bounding box.
- `BB.corners : BB.bbox -> Plane.point * Plane.point * Plane.point * Plane.point`
  Returns the four corners of the bounding box in order

  (top left, top right, bottom left, bottom right)

2.3 Barnes-Hut

In lecture, we discussed how to solve the n-body problem in the naïve, quadratic manner. The code for this is given in `mechanics.sml` and `naiveNBody.sml`. Recall that the pieces of information we need about a body in space are its mass, location, and velocity. This is represented by the type definition

```
type body = Plane.scalar * Plane.point * Plane.vec
```

The type `body` is used to represent the different bodies in the n-body simulation. Specifically, in an expression `(m, p, v)` of type `body`, `m` is the mass of the body, `p` is its position, and `v` is the vector representing its velocity.

The naïve, quadratic implementation of an n-body simulation is given by the function

```
accelerations : body Seq.seq -> Plane.vec Seq.seq
```

in `naiveNBody.sml`. This function transforms a sequence of bodies into a sequence in which the element at position `i` represents the acceleration for the element at position `i` of the sequence of bodies.

One of the vital helper functions for this is

```
accOn : body * body -> Plane.vec
```

found in `mechanics.sml`. Recall the specification is that `accOn (b1, b2)` calculates the acceleration on `b1` due to `b2`. Using this function, the calculation is fairly straightforward:

```
fun accelerations (bodies : body Seq.seq) : Plane.vec Seq.seq =
    Seq.map (fn b1 => Plane.sum (fn b2 => accOn (b1, b2)) bodies) bodies
```

However, on large inputs, this implementation is accurate, but unacceptably slow for an actual simulation. There are many different approximations that have been developed; the one we will look at is called Barnes-Hut.

2.3.1 The algorithm

In short, Barnes-Hut groups bodies by quadrants (in the two-dimensional case) and uses a threshold value θ to determine whether each individual body is "far enough" away from a group of other bodies. If it is, it groups the other bodies into a big *pseudobody* and uses that for the acceleration calculation instead of each individual body composing the pseudobody. This results in a loss of accuracy, but a dramatic speedup in terms of runtime—while the old algorithm had work in $O(n^2)$, this algorithm's work is in $O(n \log n)$ if the threshold value is well-chosen.

To calculate the effect of a pseudobody on another body, it is important to know the total mass of all the bodies represented by the pseudobody and also their center of mass or *barycenter*. Therefore, when we form a pseudobody, we will compute a tuple `(m, c)` such that `m : Plane.scalar` is the total mass of the bodies and `c : Plane.point` is the barycenter. To compute the barycenter, we compute a weighted average of the vectors corresponding to the displacement of each body's position from the origin. For example, if the positions are given by the set $\{p_i \mid i \in I\}$ and the corresponding masses are given by the set $\{m_i \mid i \in I\}$ then we compute the following vector:

$$\mathbf{R} = \frac{\sum_{i \in I} m_i \mathbf{r}_i}{\sum_{i \in I} m_i}$$

where $\mathbf{r}_i$ is the vector corresponding to the displacement of position p_i from the origin. The barycenter is then the head of this vector.

Given the total mass and barycenter, we approximate the acceleration due to all the bodies in the group as the acceleration due to a single body located at the barycenter with mass equal to the total mass.

2.3.2 Computing the barycenter

Rather than computing the barycenter for each quadrant of space from all of the points that space, we will approximate the barycenter as an average of the averages of the four subquadrants of a region of space. This means that all we ever need to do is to compute the barycenter of four pairs of masses and points.

Task 2.1 (10 pts). Write the function

```
fun barycenter ((m1,p1) : (Scalar.scalar * Plane.point),
                 (m2,p2) : (Scalar.scalar * Plane.point),
                 (m3,p3) : (Scalar.scalar * Plane.point),
                 (m4,p4) : (Scalar.scalar * Plane.point)) :
                Scalar.scalar * Plane.point = ...
```

that computes the pair `(m, c)` where `m` is the total mass of the four bodies (*i.e.*, the sum of the first components of the pairs) and `c` is the barycenter.

2.3.3 Grouping bodies

We still have not discussed exactly *how* to group bodies. There are many different ways of doing so, but the most straightforward is by grouping things into quadrants (for the 2D case). That is, starting at the center of the area, we divide the field into quadrants, then recursively group the bodies in each quadrant, stopping when a region has either zero or one body in it. This yields a tree-structured division of space, where each node has four subtrees, corresponding to the four quadrants of it. We can represent this tree structure as a datatype in SML:

```
datatype bhtree =
    Empty
  | Single of body
  | Cell of (Scalar.scalar * Plane.point) * BB.bbox
            * bhtree * bhtree * bhtree * bhtree
```

`Empty` represents a region with no bodies in it. `Single b` represents a region with exactly the body `b` in it. `Cell ((m, c), bb, sq)` is somewhat more complicated:

- `m` is the total mass of the bodies contained in the region.
- `c` is the barycenter of the bodies contained in the region.
- `bb` is a bounding box of the region.
- The four subtrees represent the subdivisions of the four quadrants of the region. The four child `bhtree`'s are, in order, the top-left, top-right, bottom-left, and bottom-right quadrants of the region, respectively.

As a first step in constructing this tree, we will write the `quarters` function to split a bounding box into four equally sized quadrants.

Task 2.2 (10 pts). Write the function

```
quarters : BB.bbox -> BB.bbox * BB.bbox * BB.bbox * BB.bbox
```

to compute the four bounding boxes that correspond to the top-left, top-right, bottom-left, and bottom-right quadrants of the argument bounding box. Use the bounding box functions described above.

2.3.4 Growing the tree

We now have the tools we need to compute a `bhtree` from a sequence of points and a bounding box.

Task 2.3 (30 pts). Write the function

```
compute_tree : body Seq.seq * BB.bbox -> bhtree
```

such that `compute_tree (s, bb)` evaluates to `T`, where `T` is the tree decomposition of `s` in the bounding box `bb`. You may assume that all of the bodies in `s` are within the bounding box `bb` and that no two bodies in `s` occupy the same position (*i.e.*, have equal position components).

In the recursive calls, you will need to divide `s` into four sequences corresponding to those bodies in each of the four quadrants of `bb`. If a body is on the border between two quadrants, it should be placed in the *first* quadrant that it is in, in the following order: top left, top right, bottom left, bottom right.

Note: The barycenter of the bodies in a bounding box should be computed as the barycenter of the four quadrants' barycenters. You can use the helper function `center_of_mass : bhtree -> Plane.scalar * Plane.point` to project the relevant data from the result of a recursive call.

2.3.5 Computing acceleration

Now that we can calculate the tree determined by a group of bodies, we can use it to efficiently compute an approximation of the acceleration of all the bodies at this particular timestep. This brings us back to the threshold value θ mentioned above.

The reason Barnes-Hut is more efficient than the naïve approach is that it does not compute the exact acceleration—instead, it uses a parameter θ to determine exactly how precise to be. Whenever your algorithm reaches a region with more than one body in it (that is, a `Cell` in the tree), it checks to see if $\frac{diam}{dist} \leq \theta$, where `diam` is the diameter of the region (the length of the diagonal) and `d` is the distance from the body being checked to the region's barycenter. If it is, then the region is treated as one large body located at its barycenter (which we have conveniently already calculated!). Otherwise, the region gets decomposed into quadrants and the respective accelerations from the bodies in each quadrant are computed recursively, combined, and returned.

Task 2.4 (5 pts). Write the function

```
val too_far : Plane.point * Plane.point * BB.bbox -> Scalar.scalar -> bool
```

such that given a point `p1` (position of body whose acceleration is being computed), a point `c` (center of a region), and bounding box `bb` (bounding box of that region) and a threshold `t` (θ), `too_far (p1, c, bb, t)` evaluates to `true` if $\frac{diam}{dist} \leq \theta$ and `false` otherwise.

Task 2.5 (25 pts). Write the function

```
bh_acceleration : bhtree * Plane.scalar * body -> vec
```

such that `bh_acceleration (T, threshold, b)` computes the acceleration on `b` from the tree `T` according to the algorithm described above. (**Hint:** Use a function from `mechanics.sig`, which is the mechanics code from lecture.)

Task 2.6 (20 pts). Finally, write a function

```
    barnes_hut  : Scalar.scalar * body Seq.seq -> vec Seq.seq
```

that uses your `compute_tree` and `bh_acceleration` functions to form the Barnes-Hut tree for a sequence of bodies and then use it to compute the acceleration on each body in the sequence.

2.4 How to Load the Project

To load your code using the floating point implementation of the plane, in SMLNJ issue the command

```
- CM.make "sources-real.cm";
```

To load your code using the rational implementation of the plane, in SMLNJ issue the command

```
- CM.make "sources-rat.cm";
```

`CM.make` uses the SMLNJ *compilation manager* to load many different files, including all of our support code and the code that you write. **Every time you edit your file, you should re-run `CM.make` to reload your code—do this in place of "using" the homework file.**

2.5 Testing

Barnes-Hut is more difficult to test than the previous homeworks, because the data structures are more complex and examples are harder to write out by hand. We encourage you to test your code using all of the following techniques:

2.5.1 Unit Tests

Because your solution consists of several functions that build on each other, it is in your interest to test each function in isolation, which will help you figure out where your bugs are. For `barycenter`, `quarters`, and `compute_tree`, we have provided some tests, and you can write more (if you want) using the helper functions and hard-coded points/bounding boxes/bodies in the module `TestData`. You can run `BarnesHut.test_X()` for the `test` functions in `BarnesHut.sml`.

2.5.2 Testing Exact Answers with the Rational Plane

For the final Task 5.6, you should test your overall implementation by running on the rational plane and `diff`ing the output against ours, as described below. You do not need to include hard-coded tests in your SML file. Running the visualization with the output from the floating point scalars should give you a rough idea of if your code is correct, but comparing with our output on the rational scalars will be more precise.

We have provided the output of our code on the rational scalars on a few examples. These are in the `example-transcripts` directory.

You can compare your output with ours as follows:

```
- CM.make "sources-rat.cm";
- Transcripts.run_tests();
(exit SMLNJ)
$ diff -q tests/ expected-transcripts/
```

If you see nothing, that means the output was what it should be. If you see lines like

```
Files tests/rat.system.2day.auto.txt
and expected-transcripts/rat.system.2day.auto.txt differ
```

that means your output was different than ours in that case. You can ignore output about `.DS_store` (a MacOS file), and if you see output about `real.system.*.*.txt` only being in the `tests` directory that just means that you accidentally ran this tester with the real plane instead of the rational one.

If the tests fail, running the solar system transcripts (described next) in the visualizer (described at the end) might be helpful for debugging, since it will give you a visual sense of whethwer the simulation is mostly right or not.

2.5.3 Generating Transcripts

Another way to test is to generate transcripts and use the visualizer to see what they look like. A good first goal is to see that the planets orbit the sun:

- `Transcripts.run_solar_inner : int * string -> unit`
  `run_solar_inner (days, outfilename)` generates a transcript for running the solar system for that many days and puts the results in the file `data/outfilename`. The transcript file tells the visualizer to show only the planets up to Mars.

  For example:

  ```
  - Transcripts.run_solar_inner (365, "year.txt.sim");
  ```

  will create a file `data/year.txt.sim` that should show the earth orbiting the sun once.

- `Transcripts.run_solar : int * string -> unit` Same as the above, except the visualization radius includes all planets.

Next, you should run some larger simulations:

- `Transcripts.run_file : string * int * (IntInf.int * IntInf.int) -> unit`

  `run_file (filename, num_iters, timestep)` runs the simulation on an input file specified by `filename`, for `num_iters` steps, with time given by `Scalar.fromRatio timestep`. The output is written to `filename.sim`.

  For example:

  ```
  - Transcripts.run_file ("data/galaxy2.txt", 2000, (1,10));
  ```

produces a file `data/galaxy2.txt.sim`. See `data/datafiles.txt` for descriptions of the simulations.

Here are some good timesteps and numbers of iterations.

```
Transcripts.run_file ("data/asteroids1000.txt", 1000, (1,10)); (* 62 seconds*)
Transcripts.run_file ("data/cluster2582.txt", 2000, (1,10)); (* 555 seconds *)
Transcripts.run_file ("data/galaxy1.txt", 2000, (1,10)); (* 130 seconds *)
Transcripts.run_file ("data/galaxy2.txt", 2000, (1,10)); (* 83 seconds *)
Transcripts.run_file ("data/galaxy3.txt", 1500, (1,10)); (* 304 seconds *)
Transcripts.run_file ("data/galaxy4.txt", 2000, (1,10)); (* 41 seconds *)
Transcripts.run_file ("data/spiralgalaxy.txt", 2000, (1,10)); (* 94 seconds *)
Transcripts.run_file ("data/galaxymerge1.txt", 5000, (1,5)); (* 820 seconds *)
Transcripts.run_file ("data/galaxymerge2.txt", 2500, (1,10)); (* 626 seconds *)
Transcripts.run_file ("data/galaxymerge3.txt", 2500, (1,10)); (* 655 seconds *)
Transcripts.run_file ("data/galaxyform2500.txt", 2000, (1,10)); (* 294 seconds *)
Transcripts.run_file ("data/collision2.txt", 2500, (1,10)); (* 330 seconds *)
Transcripts.run_file ("data/collision1.txt", 1500, (1,10)); (* 299 seconds *)
Transcripts.run_file ("data/saturnrings.txt", 100, (1,100)); (*  112 seconds *)
Transcripts.run_file ("data/galaxy10k.txt", 100, (1,10)); (* 93 seconds *)
Transcripts.run_file ("data/galaxy20k.txt", 50, (1,10)); (* 188 seconds *)
Transcripts.run_file ("data/galaxy30k.txt", 800, (1,10)) (* 1736 seconds *)
```

Some of them take a while (the comment is how long they took for me). You don't need to run all of them — you can pick a few to try. Or you can either turn down the number of iterations to see less of the movie, or run them overnight. The total size of all files produced is about 2GB.

- The command

  ```
  - Transcripts.run_files();
  ```

  runs all of the above.

2.5.4 Running the Visualizer

Once you have produced a transcript file, you can visualize it by navigating to

```
https://dlicata.wescreates.wesleyan.edu/teaching/fp-f22/visualizer/visualizer.html
```

You can then load a transcript file in one of two ways: either dragging and dropping the transcript file into the dashed box, or using the file browser to select the file manually. **The visualizer will only work with floating point transcripts!** Once you select a transcript, click 'Go!' to run the visualizer. You should refresh the page before running another transcript.