15-150 Fall 2013
Homework 10

Out: Wednesday, 6 November 2013
Due: Tuesday, 12 November 2013 at 23:59 EST

1 Introduction

In this homework, you will see two simple implementations of a sequence library, and later implement the Barnes-Hut approximation algorithm for the n-body problem.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

Important: for this assignment, we are testing a new and improved submission system. To submit your solutions, run `make` from the `hw/10` directory. Now, however, instead of download/uploading a `.tar` file yourself, the script will automatically upload the `hw10.tar` to the Autolab servers. If this new system should fail, then follow the instructions in the file `IN_CASE_OF_EMERGENCY`.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your PDF is valid; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

You should have a file `hw10.pdf` containing your written solutions. Your `barnes-hut.sml`, `shrubseq.sml`, `sizeseq.sml`, and `tester.sml` files must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 12 November 2013 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format

```
val <return value> = <function> <argument value>
```

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

2 Tree Sequences

In lecture we covered an abstract data structure called Sequences, which you made use of in the last assignment (and will continue to use for the rest of the course). So far, we have avoided discussing implementations of a Sequence library; thanks to representational independence[1], the implementation details are not important for most of the code you will write in this class. The next few problems will give you a brief introduction to implementing sequences using a familiar data structure: binary trees!

We provide the following (shortened) signature in the file **tseq.sig**:

```
signature TSEQ =
sig
    type 'a seq
    exception Range
    val length : 'a seq -> int
    val nth : int -> 'a seq -> 'a
    val tabulate : (int -> 'a) -> int -> 'a seq
end
```

These are the only functions on sequences you will use for this part of the assignment. You should not use the Seq library functions, nor implement any functions outside this signature (but keep them in mind for the second part of the assignment, Barnes-Hut!)

2.1 Sequential Shrubberies

We can implement the above signature using a variety of underlying data structures; with correct code, the only difference for a user should be the run-time of our functions. In the file **shrubseq.sml** we have provided the beginnings of a sequence implementation using the following datatype:

```
datatype 'a sseq = Empty
                 | Leaf of 'a
                 | Node of 'a sseq * 'a sseq
type 'a seq = 'a sseq
```

For a type `t`, a value of type `t seq` is a binary tree with data of type `t` stored at the leaves. Additionally, we have the following *balance invariant*: For any `Node(L,R)` the number of leaves in `L` and in `R` differ by at most 1. Your code must ensure that this invariant is always maintained. Note that since we do not support an `insert` or `delete` operation on sequences, it is enough to guarantee that our trees are balanced when we first create them.

[1] Recall that *representational independence* is the idea that we can use multiple implementations of a data structure to represent the same signature, in this case ordered lists of elements (sequences).

Task 2.1 (2%). We have provided an implementation of the `length` function - it should look very familiar from your previous work with binary trees. What is the work and span of `length`? (Hint: It's not $O(1)$). Briefly explain why.

Task 2.2 (10%). Implement the `nth` function. Your implementation must use `length`, but NO other helper functions, and will have work in $O(n)$ and span in $O((\log n)^2)$. Remember that sequences are indexed beginning at zero; your code should raise a `Range` exception if the argument index is invalid for the given sequence.

Note: There is a way to implement `nth` in $O(n)$ work and $O(\log n)$ span using a helper function and certain assumptions about the underlying tree—you should **not** write this implementation. Your code for this task should not be very long or complex; in the next section, you will see an alternative optimization that makes this simple implementation more efficient.

Task 2.3 (10%). Implement the `tabulate` function. DO NOT use `length`, `nth`, or any other helper functions. Your implementation must have work in $O(n)$ and span in $O(\log n)$ (you may assume that the input function has $O(1)$ work and span). Remember to ensure that your trees are balanced!

Do not forget to test your implementation; you can test your functions together by using `tabulate` to create a simple sequence, then checking for correct values with `length` and `nth`.

Clearly, the above implementation is not ideal; your answer to the first question above should give you a clue as to why. There are several ways to improve our implementation; in the next task, you will do exactly that, focusing on our implementation of `length`.

2.2 Sizable Shrubberies

We would prefer for our sequences to implement `length` with constant work and span. One simple way to achieve this is to modify our underlying datatype. In the file **sizeseq.sml** we provide the following datatype:

```
datatype 'a sseq = Empty
                 | Leaf of 'a
                 | Node of 'a sseq * int * 'a sseq
type 'a seq = 'a sseq
```

As before, for any type `t` a value of type `t seq` is a *balanced* binary tree with data stored at the leaves. This time, however we store an additional value of type `int` in each *node*, with the following invariant: For every `Node(L,s,R)` the value of `s` equals the size of the subtree rooted at this node (i.e. the number of leaves). In other words, we have augmented our tree to store the length of the sequence at the root node (for sequences of length at least two), and similarly the lengths of corresponding subsequences at child nodes. This leads to some clear improvements:

Task 2.4 (2%). Implement `length` in **sizeseq.sml**; your implementation must have $O(1)$ work and span.

Task 2.5 (1%). Copy your implementation of `nth` from **shrubseq.sml** to **sizeseq.sml**; it should work with only minor modification. As mentioned in lecture, your implementation should now have work and span in $O(\log n)$.

From the last two problems, you should see how an improvement in one part of the code can lead to simple yet fast code in other areas. However, this comes at a slight trade-off.

Task 2.6 (2%). Implement `tabulate` in **sizeseq.sml**; you should be able to re-use most of your code from the previous section. However, you must now ensure that our second invariant (storing the size in each node) is satisfied!

Task 2.7 (3%). Do the work and span of `tabulate` change, and if so, what are the new O-bounds? Briefly explain why, and describe the cost trade-off that occurs between `tabulate` and `length`.

Remember to test your implementations of `length`, `nth`, and `tabulate` as before.

These two implementations show how we can achieve $O(1)$ time for `length` and a reasonable run-time for some of our functions using balanced trees, but in other cases we are restricted by our choice of data structure. Using alternative implementations with arrays or vectors (like the `Seq` library you will use for Barnes-Hut) we can actually achieve $O(1)$ work for `nth` and span for `tabulate`. In this class we will focus primarily on *using* sequences for functional programming, but if you are interested you can learn more about this type of problem through further courses in data structures and algorithms such as 15-210.

3 Barnes-Hut

For the remainder of this assignment, you will be implementing the Barnes-Hut algorithm for efficient n-body simulation, an extension of the ideas you learned in lecture.

3.1 Preface

The following points explore the various libraries you will be using for this homework. Please read through each section to ensure you fully understand the code infrastructure for the coming tasks.

3.1.1 The plane problem

For n-body simulations, we need a representation of numbers (or scalars) to talk about distance between bodies in a 2D plane. We have two options for numerical representation:

1. **Reals.** Reals are fast to compute and work well for visualization, but alas they are only 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
   ```

   Hence, reals make it difficult to test our code for correctness.

2. **Rationals.** Rationals (or all numbers which we can represent as a fraction $\frac{a}{b}$ where $a, b \in \mathbb{Z}$) are not built in to SML but give us a precise representation for non-integer numeric values. Consequently, rationals make testing more accurate than if we used reals.

   Computation on rationals, however, is slow because we have to do arbitrary precision integer arithmetic. Moreover, we cannot compute Euclidean distance (i.e. in a 2D plane) with rationals. If x and y are rational numbers, it is not necessarily the case that $\sqrt{x^2 + y^2}$ is a rational number. We can, however, compute the Manhattan distance between points[2].

[2] http://en.wikipedia.org/wiki/Taxicab_geometry

3.1.2 The plane solution

Rather than choose one or the other, we will instead parameterize our Barnes-Hut algorithm by giving it an implementation of numbers inside a package called `Plane`. `Plane` contains a number of useful functions and types, including `Plane.scalar`, `Plane.point`, and `Plane.vec` for scalars, points, and vectors, respectively.

We have provided to you two implementations of `Plane` using reals and rationals which you can find in `lib/realplaneargs.sml` and `lib/ratplaneargs.sml`, respectively. The file `lib/space.sig` lists all of the functions common to both implementations of the plane. The `lib/makeplane.sml` and `lib/makescalar.sml` also have some common functions to the `Plane` signature which you may find useful.

3.1.3 Scalar library

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`
  `Scalar.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; see `lib/scalar.sig` for a description. By using only the above operations, your code will work with either implementation of the plane.

3.1.4 Point and vector libraries

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. Whatever the definition of `Scalar.scalar` is, 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. Note you may see the type `Plane.scalar` while debugging - this is defined to be the same as `Scalar.scalar`. Here we give a brief description of some of the functions you may need on this assignment:

- `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.
- `Plane.sum : ('a -> Plane.vec) -> 'a Seq.seq -> Plane.vec`
  `Plane.sum f s` evaluates to the vector that corresponds to the sum of the sequence of vectors that results from mapping `f` on `s`. This corresponds to the mathematical notion of a sum of vectors, `sum f` $\langle s_1, \ldots, s_n \rangle = \sum_{i=1}^{n} f(s_i)$.

3.1.5 Other libraries

There is a lot more starter code for this assignment than for any previous assignment. Only edit the files `barnes_hut.sml` and `tester.sml`. When writing your solution, you should use functions specified in:

- `lib/bbox.sig`
- `lib/space.sig`
- `lib/scalar.sig`
- `lib/mechanics.sig`
- `../../../src/sequence/sequencecore.sig`
- `../../../src/sequence/sequence.sig`

3.1.6 Testing

You have two methods for testing your code:

- If you want to check your code for correctness, you will use the rational implementation of the `Plane` (see Section 3.6.1).
- If you want to visualize your n-body simulation, you will use the real implementation of the `Plane` (see Section 3.6.2).

3.2 Naïve simulation

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.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 naive implementation that we gave in lecture has been rewritten using planes and the following 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 `i`-th body.

However, this implementation is accurate on large inputs but unacceptably slow for an actual simulation. You will implement a more efficient approximation to find accelerations.

3.3 The Barnes-Hut algorithm

Barnes-Hut groups bodies into quadrants 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. 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 : Scalar.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 point that results from displacing the origin point by $\mathbf{R}$.

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.

3.3.1 Computing the barycenter

We will begin by writing the `barycenter` function to compute the total mass and barycenter of a sequence of pairs of masses and points.

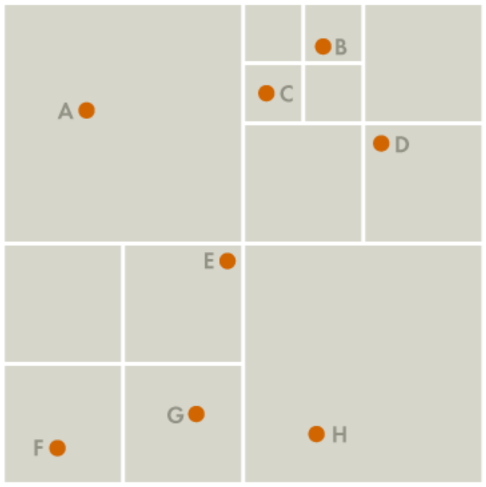
Task 3.1 (10%). Write the function

```
barycenter : (Scalar.scalar * Plane.point) Seq.seq
           -> Scalar.scalar * Plane.point
```

that computes the pair `(m, c)` where `m` is the total mass of the bodies in the sequence (*i.e.*, the sum of the first components of the pairs) and `c` is the barycenter. You should use the provided `scale_point` function to compute a weighted vector from a mass and position. You should also use the `Plane.sum` function.

3.3.2 Grouping bodies

To group bodies, we will partition them into quadrants. Starting at the center of the plane we are considering, 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. Here's a sample division of a set of bodies[3]:



Note: this diagram does not precisely represent what your BH tree looks like, because although we use quadrants for dividing the bodies, we group the bodies within a quadrant by their bounding box and then subdivide that box.

This process yields a tree-structured division of space with the following datatype:

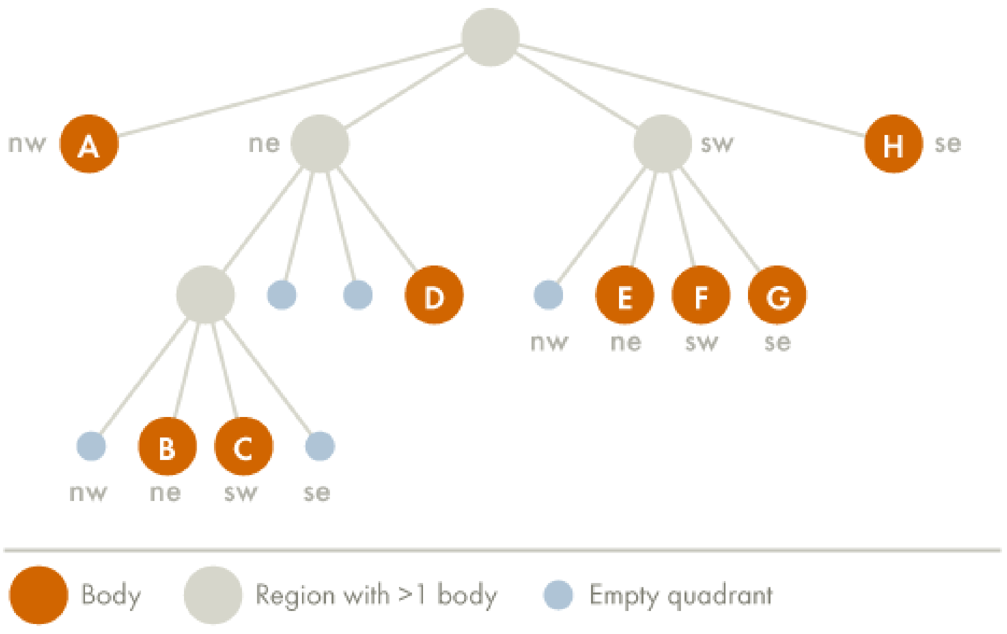
```
datatype bhtree =
    Empty
  | Single of body
  | Cell of (Scalar.scalar * Plane.point) * Scalar.scalar * (bhtree Seq.seq)
```

[3]Credit: http://arborjs.org/docs/barnes-hut

`Empty` represents a region with no bodies in it. `Single b` represents a region with exactly the body `b` in it. For `Cell ((m, c), bd, sq)`, we have:

- `m` is the total mass of the bodies contained in the region.
- `c` is the barycenter of the bodies contained in the region.
- `bd` is the diameter of the bounding box of the region. A bounding box for a set of bodies is a rectangular region which contains all the bodies in the set. Some functions will reference the type `BB.box` which represents the bounding box, and you will find functions related to the bounding box in `lib/bbox.sig`. For our tree, however, you will only store the diameter of the bounding box, not the box itself.
- `sq` is the sequence of the four quadrants in the region. The invariant is that this sequence is always of length four and that the four child `bhtree`'s are, in order, the top-left, top-right, bottom-left, and bottom-right quadrants of the region, respectively.

So the above example would have a tree that looks like:



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 3.2 (10%). Write the function

```
quarters : BB.box -> BB.box Seq.seq
```

to compute a sequence of four bounding boxes that correspond to the top-left, top-right, bottom-left, and bottom-right quadrants of the argument bounding box, in that order. You may find some of the functions in `lib/bbox.sig` helpful. In particular, note that the `BB.vertices` function computes the sequence of the top-left, top-right, bottom-left, and bottom-right corners of a bounding box.

3.3.3 Growing the tree

Once we have the four quadrants, we need to partition the bodies between them. To do this we will use the function

```
take_and_drop : BB.box -> body Seq.seq -> (body Seq.seq * body Seq.seq)
```

The expression `take_and_drop bb s` evaluates to a pair of sequences in which all of the bodies in the first box are `BB.inside` of `bb`, and the bodies in the second sequence are not.

Observe that we will need to be careful when using `take_and_drop` to partition the bodies in a bounding box we desire. You should break ties (*i.e.*, points that are on the line dividing two quadrants) in favor of the first of the two quadrants in the sequence.

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

Task 3.3 (20%). Write the function

```
compute_tree : body Seq.seq -> bhtree
```

such that `compute_tree s` evaluates to `T`, where `T` is the tree decomposition of `s`. You may assume that no two bodies in `s` occupy the same position (*i.e.*, have equal position components). Your implementation does not have to handle arguments that violate this condition. **You must use `Seq.map` to compute the recursive calls in parallel!**

Note 1: you will need to compute the bounding box for the given bodies and store its diameter inside `Cells` of your tree. We have provided you a few helper functions which you may find useful for this task. First, you have

```
position : body -> Plane.point
```

which returns the position of a body. Second, you have

```
BB.hull : Plane.point Seq.seq -> BB.box option
```

which given a sequence of points returns a bounding box containing all points in the sequence (`BB.hull` returns `NONE` if the input sequence is empty). Lastly, you have

```
BB.diameter : BB.box -> Scalar.scalar
```

which given a bounding box returns the diameter of the bounding box.

Note 2: If a large set of bodies is partitioned into several subsets and the barycenter of each subset is known, the barycenter of the whole set can be more efficiently computed as the barycenter of the barycenters of the subsets. In this instance, the barycenter of the bodies in a bounding box can be computed as the barycenter of the four quadrants' barycenters. You should use this observation to compute the barycenter in the recursive case of `compute_tree` by applying the `barycenter` function you wrote earlier to the sequence of the barycenters of the quadrants given by the results of the recursive calls. You may find the function `center_of_mass : bhtree -> Scalar.scalar * Plane.point` helpful to project the relevant data from the sequence of recursive results.

3.3.4 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 to the n-body problem is that it does not compute the exact acceleration—instead, it uses 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 $\frac{d}{r} \leq \theta$ (where `d` is the diameter of the region's bounding box and `r` 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 3.4 (10%). Write the function

```
too_far : Plane.point -> Plane.point -> Scalar.scalar -> Scalar.scalar -> bool
```

such that given a point `p`, a location `c` of a pseudobody, a diameter `bd` for the pseudobody's bounding box and a threshold `t`, `too_far p c bd t` evaluates to `true` if $\frac{d}{r} \leq \theta$ and `false` otherwise. Recall that in this equation, d refers to the diameter of the pseudobody's bounding box, r refers to the distance between the point and the pseudobody, and θ corresponds to the threshold argument `t`. You may find the function `Plane.distance` useful.

Task 3.5 (20%). Write the function

```
bh_acceleration : bhtree -> Scalar.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:** The function `accOnPoint` in `lib/mechanics.sig` may be useful.) **You should use `Plane.sum` to compute the recursive calls in parallel and add the resulting accelerations!**

The provided function `barnes_hut` 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.

3.4 Testing your code

Now that you have written all of the important code, you can actually visualize an n-body simulation with some of the infrastructure that we have given you!

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. **Although you should take advantage of the visualizer and transcripts, you must also write your own tests for Tasks 3.1-3.4 in tester.sml**. You will also find some utilities to help you test in `tester.sml`.

- Run the visualizer! The existence of many bugs will be pretty easy (and funny) to spot.
- You can use the functions provided in `tester.sml`. This might be especially helpful for debugging particular functions. In it you will find functions to test for equality of sequences, boxes, and much more.
- For the final task 3.5, 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 for this one task 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.

Note that the Barnes-Hut algorithm that you implement is not the same as the naïve quadratic algorithm! The former is an approximation of the latter. Therefore, you should expect the visualization of the output of your code to be roughly the same as the visualization of the output of the naïve code, but the outputs themselves will not agree.

3.5 Compiling your code

To load your code, at the REPL issue the command

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

3.6 Generating transcripts

Once you have done the `CM.make`, you can use the functor `Transcripts` in `transcripts.sml` to generate transcipts. A transcript is a file in which each line represents the position of a body at a time step. Using `Transcripts` produces many example transcripts, but if you want to make your own simulations, you can write your own functor to produce transcripts for your own universes. In particular, the visualizer is a lot more fun if you simulate the solar system for a year, but we left this out because generating the rational version of the transcript takes too long.

3.6.1 Rational transcripts

You can use a rational plane transcript to test the correctness of your code.

To create rational plane transcripts, you can type the following commands into the REPL:

```
- CM.make "sources.cm";
- RatTrans.go();
```

This will create a number of files that all end in `auto.txt`, and may take a few minutes. You can safely delete these files at any time.

We have provided you with a number of rational transcripts in the folder `tests`. The transcripts that you generate in this way should match the transcripts that we have provided exactly. To compare your transcripts with ours, use the UNIX utility `diff`. For example,

```
  diff rat.onebody.1day.auto.txt tests/rat.onebody.1day.auto.txt
```

If `diff` doesn't make any output, as in

```
  bovik@unix34 hw10 % diff rat.onebody.1day.auto.txt tests/rat.onebody.1day.auto.txt
  bovik@unix34 hw10 %
```

then the files are exactly the same and your code agrees with ours on that case. If `diff` does produce output, as in

```
  bovik@unix34 hw10 % diff rat.onebody.1day.auto.txt tests/rat.onebody.1day.auto.txt
  2c2
  < (0, 0),(0, 57909100000),(0, ~108208930000),(0, ~149600000000),(0,
  227939100000),(0, 778547200000),(0, ~1433449370000),(0, 2876679082000),(0,
  ~45)
  ---
  > (0, 0),(0, 57909100000),(0, ~108208930000),(0, ~149600000000),(0,
  227939100000),(0, 778547200000),(0, ~1433449370000),(0, 2876679082000),(0,
  ~4503443661000)
  bovik@unix34 hw10 %
```

then your code does not agree with ours on that case.[4]

We've provided a script `comparetranscripts.sh` that will help you do this as well. If your code is correct, the output produced by running `comparetranscripts.sh` should look like this:

```
  bovik@unix34 hw10 % comparetranscripts.sh
  checking rat.onebody.1day.auto.txt against tests/rat.onebody.1day.auto.txt
  checking rat.onebody.1yr.auto.txt against tests/rat.onebody.1yr.auto.txt
```

[4]`diff` is a robust and useful UNIX utility for comparing text; you can learn more about it starting at http://en.wikipedia.org/wiki/Diff.

```
checking rat.onebody.2weeks.auto.txt against tests/rat.onebody.2weeks.auto.txt
checking rat.system.1day.auto.txt against tests/rat.system.1day.auto.txt
checking rat.system.2day.auto.txt against tests/rat.system.2day.auto.txt
checking rat.twobody.1day.auto.txt against tests/rat.twobody.1day.auto.txt
checking rat.twobody.2day.auto.txt against tests/rat.twobody.2day.auto.txt
checking rat.twobody.3day.auto.txt against tests/rat.twobody.3day.auto.txt
checking rat.twobody.4day.auto.txt against tests/rat.twobody.4day.auto.txt
checking rat.twobody.5day.auto.txt against tests/rat.twobody.5day.auto.txt
checking rat.twobody.6day.auto.txt against tests/rat.twobody.6day.auto.txt
bovik@unix34 hw10 %
```

3.6.2 Real transcripts

You can use a real plane transcript to run the visualizer.

To create real plane transcripts, you can type the following commands into the REPL:

```
- CM.make "sources.cm";
- RealTrans.go();
```

As stated above, calculations with reals are much faster, and the transcipts should be created almost instantly. All the transcripts end in `auto.txt` and you can safely delete any of these files.

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

```
http://www.cs.cmu.edu/~15150/visualizer/
```

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 real transcripts!** Once you select a transcript, click "Go!" to run the visualizer.

In the visualization, objects are displayed as circles. Note that the scale of the display will be chosen automatically so that all bodies are visible on-screen at all times. If one body is very far away from the others at some step of the simulation, this can result in objects at different positions appearing very close or even being in distinguishable in the visualization.

Note: running `RealTrans.go()` will generate more transcripts of greater complexity than using the rational transcripts. Try putting `real.system.1001day.auto.txt` into the visualizer for interesting results.