Term Project Report

Paul English

July 10, 2014

1 Mathematical Approach

For solving the systems in the satellite and receiver programs I made use of Newton's method. It's straightforward to understand and made for a simple implementation. In working out the mathematical models of the program beforehand it greatly simplified implementation. It served as a solid specification and resource for knowing how to code the project, and how to ensure validity. For any numerical project, there is no doubt that it is essential to first understand the math and then the implementation will be trivial.

2 Software

I made use of Clojure to implement the program, which is a functional lisp built to run on the JVM (Java Virtual Machine). It compiles to Java bytecode offering you platform benefits of the JVM without having to deal with the simple object-oriented requirements and limited features of the Java language. A lisp is a programming language with polish notation and lots of bananas "(() (()))" for structure. Lisp computation is based on lambda calculus, requiring only a few concepts (primarily function composition) to create turing complete programs.

2.1 Functional Programming

Clojure specifically has a few paradigms that made it very nice to use for this project. It has a functional programming style that is built around composition and abstraction. It's more preferable (as a human) to deal with small composable functions than it is to deal with low-level loops and machine details.

Particularly lazy evaluation and the sequence abstractions made for a very clean solver implementation. In Clojure we treat everything as a sequence of values. One example of this is in my solver implementation where we take a function that produces a guess, create an infinite iteration of successive guesses, and take only the first convergent guess.

```
(defn solve [start]
  (-> start
      (iterate next-guess)
      (drop-while (comp not convergent?))
      first))
```

This is not possible in a language without lazy evaluation, and ends up being very succinct do to comprehensive sequence abstractions.

2.2 Linear Algebra Routines

Additionally the ability to overload operators means we can operate on matrices similar to how you might in matlab. The following functions all operate on matrices using a library that overloads linear operations with links to the native or GPU BLAS libraries depending on what's available.

```
(sm/defn rotate-coordinates :- CartesianCoordinateList
  [times :- [BigDecimal]
   coordinates :- CartesianCoordinateList]
  (let [theta (with-precision 20 (/ (* @tau times) @s))
        rotations (emap rotation-matrix theta)
        rotated (map #(mmul %1 %2)
                     rotations
                     coordinates)]
    (parse-cartesian-list rotated)))

(sm/defn above-horizon? :- Boolean
  [vehicle :- CartesianCoordinate
   satellite :- CartesianSatelliteCoordinate]
  (> (dot (drop 2 satellite) vehicle)
     (dot vehicle vehicle)))

(defn satellite-time [{:keys [pseudorange vehicle-coordinates new-coordinates]}]
  (- (map #(distance vehicle-coordinates %) new-coordinates)
     (** pseudorange 2)))
```

Wherever possible I made use of linear operations rather than explicit loops.

2.3 Testing

Responsive testing is probably one of the more important practices to keep up. This is also a point where functional and immutable languages benefit, tests are able to deal with side-effect free small functions without having to worry about any global state or values that might change under the scenes. Throughout development I wrote a lot of tests and kept them automatically running so that I would know if I inadvertently broke something while I was changing code.

```
(fact "We can group the receiver input by the number of path points we have"
  (let [grouped-satellites (group-by-index-change input)]
    (count grouped-satellites) => 2
    (->> grouped-satellites
         first
         (map first)) => [3 4 8 11 14 15 17 20]
    (->> grouped-satellites
         second
         (map first)) => [3 4 8 11 14 15 17 20]))
```

2.4 Version Control

All software (and collaborative) projects should use version control. It keeps track of history, lets teams collaborate without stepping on toes, and ensures things don't get deleted. With just about anything that requires a lot of iterative work or work from multiple people this is a lifesaver.

```
c5673 (HEAD, master) Update tests
7500f Use threading on solver
00fbf Remove fudge factor on above-horizon? fn
d07b6 Move squared-norm fn
5270e Refactor receiver fns for clarity
dfbdf (origin/master) Add note on test checker
b224a Test all the things.
463e1 FIX the implementation of matrix->stdout
```

There are many other aspects of the software that made implementation straightforward & reviewing the code is recommended.

3 Building & Running

Obtaining the code for the project is easiest with the version control tool git. You can obtain a copy of the project by running

```
git clone http://github.com/log0ymxm/gps-sim.git
```

Which will create a local folder `gps-sim` in your current working directory.

Clojure is a language built on the JVM and any Clojure code can be run using the java command with a the right classpath, but Java classpaths can quickly become unwieldly and it's often preferable using it directly. It's common to use the Leiningen tool to manage Clojure projects easily. The following `lein` commands let you work with the project.

```
# Build jar files for use with any java install
lein with-profile vehicle:satellite:receiver uberjar
cat bm.dat | java -jar target/vehicle.jar # etc..

# Run tests
lein midje

# Interact with the code & environment
lein repl
```

We can also test and run the code from within an interactive read-eval-print-loop after running `lein repl`.

```
user> (require '[gps-sim.receiver :as receiver])
nil
user> (with-in-str (slurp "vehicle.out") (receiver/-main))
;; Receiver output...
:ok
user> (go)
;; This is an alternate way to start the tests, which
;; will retest if you change the code.
```

Making use of the repl offers the most feedback during implementation of the model. Functions can be individually tested, built, and reevaluated from disk without having to stop and start the running JVM instance.

4 Review

Overall getting an early start on the project let me play around with a few different ways of implementing this project. A functional and immutable language may not be the most efficient way to implement a simple numerical project such as this one, but the clarity and conciseness of the code is more favorable for development and understanding.

If I were to continue working on this project I may try different ways of simplifying the code. The use of Java's BigDecimal numerical type offers arbitrary-precision to avoid any arithmetic errors, but runs less efficiently and makes the code a bit more complex. The systems we're solving in this project don't require the need for arbitrary-precision, and a double or float type would work fine. Additionally it would be nice to simplify the code by making use of an automatic or symbolic differentiation library. These methods of differentiation don't have accuracy issues, and can help prevent bugs in gradient or Jacobian functions while greatly reducing the amount of code that needs to be written. Though it's important to verify and analytically calculate these functions, we need not implement them if the computer is able to do so automatically.