CS 522
Assignment 1

Jean-Sébastien Bolduc
(jseb@cs.mcgill.ca)

October 3, 2002

Causal Block Diagram Models

- Ballistic Problem
- Lorenz Equations

Exporting

- Export to LaTeX
- Export to MatLab

Time-Slicing Simulator

- Graph algorithms
- Simulator
- Results
- Study case: the ballistic problem

Requirements for the assignment are described on the assignment web page. The whole solution can be downloaded here (just untar in ATOM3.asgn).

The circle test model is given with the assignment. As required, we built a CBD for the ballistic problem. We also built a CBD for the Lorenz equations (not required — for testing purposes).

Our CBD for the ballistic problem can be found here. Whether or not this model can be simulated by your Time-Slicing simulator depends on how your simulator processes Generic blocks. The angle (block marked
theta) is in radians.

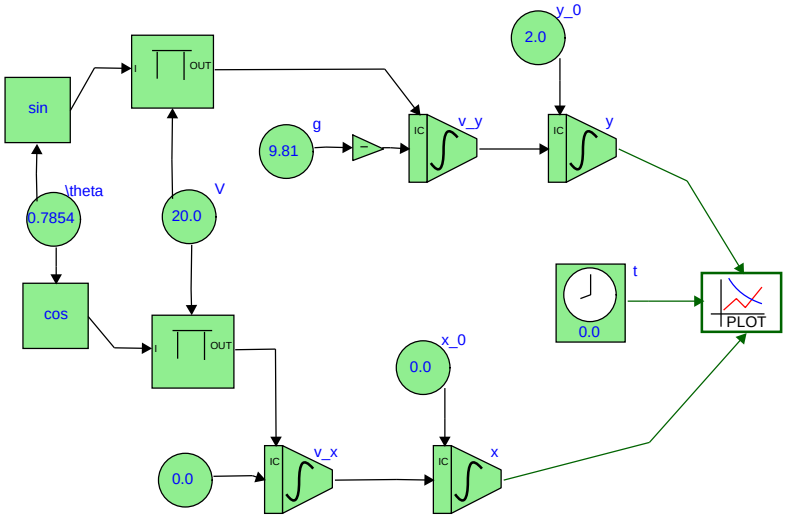

Figure 1: CBD for the Ballistic Problem

We also developped a CBD for the Lorenz Equations (this was *not* required). The model can be found here.

Algorithms to export to LaTeX or to a MatLab (or Octave) m-file are very similar: both files `EXPORT_LaTeX.py` and `EXPORT_Mfile.py` import the module `Traversal.py`.

In `Traversal.py` are 2 functions that allow to traverse the graph backward from a given connector, building a string summary of the traversal on the way. A boolean flag in those functions specifies which format we are interested in (LaTeX or m-file).

A few remarks on the exporting algorithms:

- Only CBD of ODEs can be exported, i.e., the model must have at least one integrator block.
- As opposed to the Time-Slicing simulator, the exporting algorithms do not check for the presence of algebraic loops. Note that algebraic loops *should not* be a problem when we only want to generate a LaTeX representation.
- For the algorithms to work, each integrator block must have a non empty `block_name` attribute..
- The algorithms assume—but don't verify—that the blocks' `block_name` attributes are unique.

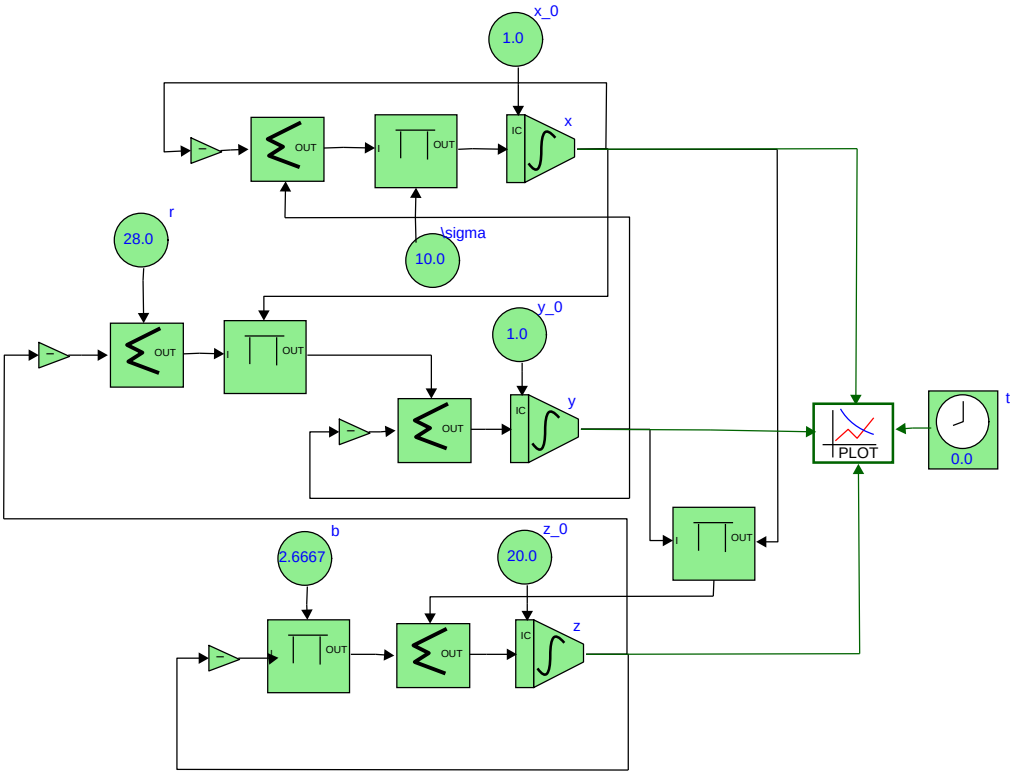

Figure 2: CBD for the Lorenz Equations

- Time blocks are not processed correctly. This is a problem only if an integrator block depends on a time block (i.e., nonautonomous systems).

The function `EXPORT_LaTeX.exportLaTeX` identifies all integrator blocks in the model. For each of these blocks, it calls `Traversal.traverse` on the input connector to build the ODE, and on the IC connector to build the initial conditions. In this case, `Traversal.traverse`'s recursion stops when either:

- Another integrator block is found
- A constant block is found
- A block with a non empty `block_name` attribute is found

In the last case, the block found is appended to a list `blockList` so it can be processed later.

Exporting the ballistic model generates the file

```
\documentclass[12pt]{article}
\begin{document}

\[
  \left\{
    \begin{array}{ll}
      \dot{v_x} = 0.0, & v_x(t_0) = v_{x,0}\\
      \dot{v_y} = -(g) , & v_y(t_0) = v_{y,0}\\
      \dot{x} = v_x, & x(t_0) = x_0\\
      \dot{y} = v_y, & y(t_0) = y_0\\
      \\
      v_{x,0} = cos(\theta)\cdot V\\
      \theta = 0.7854\\
      V = 20.0\\
      g = 9.81\\
      v_{y,0} = V\cdot sin(\theta)\\
      x_0 = 0.0\\
      y_0 = 2.0\\
    \end{array}
  \right.
\]

\end{document}
```

This gives:

$$\begin{cases} \dot{v_x} = 0.0, & v_x(t_0) = v_{x,0} \\ \dot{v_y} = -(g), & v_y(t_0) = v_{y,0} \\ \dot{x} = v_x, & x(t_0) = x_0 \\ \dot{y} = v_y, & y(t_0) = y_0 \\ \\ v_{x,0} = cos(\theta) \cdot V \\ \theta = 0.7854 \\ V = 20.0 \\ g = 9.81 \\ v_{y,0} = V \cdot sin(\theta) \\ x_0 = 0.0 \\ y_0 = 2.0 \end{cases}$$

Even though the CBD for the model does not show it, the two product blocks have a name. Removing these names in the model will have an impact on its "denotational semantics".

As in the previous case, the function `EXPORT_Mfile.exportMfile` identifies all integrator blocks in the model, and calls `Traversal.traverse` on the input connector to build the ODE, and on the IC connector to build the initial conditions. `Traversal.traverse`'s recursion stops when either:

- Another integrator block is found
- A constant block is found

To each integrator block must be assigned a unique integer that will be used as a vector index in MatLab. The dictionnary `lookUp` maps the integrator names to the corresponding indexes.

Note that the ODE solvers are different in MatLab and Octave. The m-file generated by our algorithm is to be used with MatLab. In MatLab, the function for the ODE itself *must be* in a dedicated file. The initial conditions and the initial/final times are passed as parameters to the ODE solver itself. Hence the file we produce cannot be used as it is by MatLab. We leave it as it is, because we don't want to create 2 separate files. The initial condition `x0` included in the generated file must be passed as a parameter to the ODE solver. In MatLab, the syntax is:
`>> [t x] = ode45('`*filename*`', [`*t_init t_final*`],x0);`
where *filename* is the name of the m-file. Note that the name of the function is the same as the filename (minus extension and path). For instance, generating an m-file for the ballistic problem and saving it under `ballistic.m`, we will have:

```
function xdot = ballistic(t, x)
xdot = zeros(4, 1);
xdot(1) = 0.0;
xdot(2) = -(9.81) ;
```

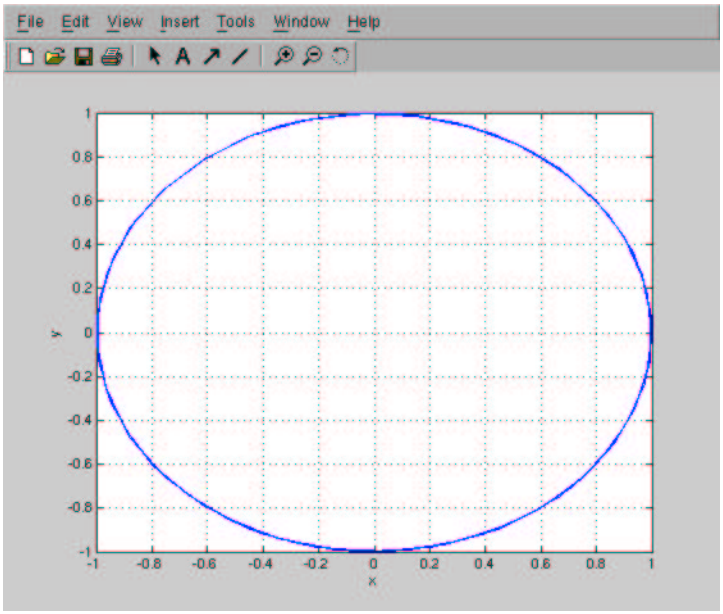

Figure 3: Circle Test

```
xdot(3) = x(1);
xdot(4) = x(2);

x0 = [cos(0.7854)*20.0; 20.0*sin(0.7854); 0.0; 2.0];
```

Pseudo-code for the time-slicing simulator was given here. Relevant files are:

`SIM_startResume.py`

`SIM_pause.py`

`SIM_reset.py`

`MODEL_plotWindow.py` (not modified)

`Graph.py`

The current version of the Time-Slicing simulator has some limitations:

- Assumes there is only one “Plot” block per model.
- No support for “File” blocks: their presence is just ignored.

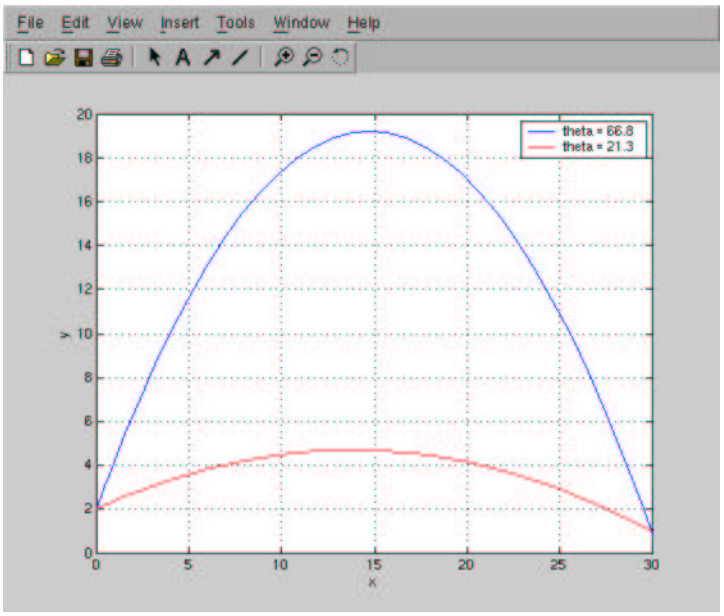

Figure 4: Ballistic Problem (optimal solutions)

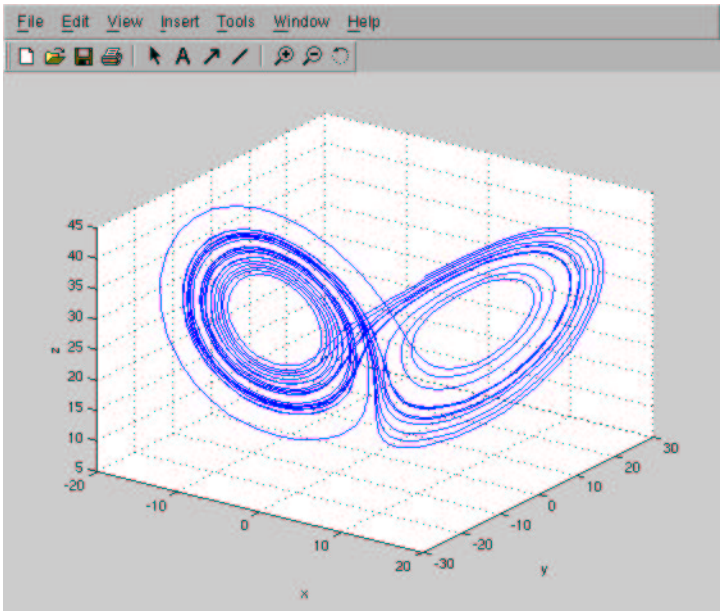

Figure 5: Lorenz Equations

- "Delay" and "Derivative" blocks are not supported: their presence in the model are interpreted as an error by the simulator.
- The current Causal-Block Diagram meta-model still allows invalid models to be constructed. For instance, a unary operator (e.g., a "Negator" block) might have many inputs. Also, a "Time" block might have an input. The current simulator makes *some* efforts to detect those errors. Ideally, the simulator should not have to be concerned with the validity of the model.
- As required, the simulator stops when it detects strongly-coupled components.

The file `Graph.py` implements both the `topSort` (topological sorting) and `strongComp` (identification of strongly coupled components) functions.

`topSort` takes as an input the list of all Block connectors, and returns the same list sorted in topological order (the *sorted dependency graph*). A few things to note:

- Since the sorted list of connectors is used throughout the simulation, it is not called from `strongComp`, but rather from the simulator.
- For the dependency graph, the connections of the original model need to be reversed (a signal on a connector depends on the signals upstream). Hence the depth-first search is performed "against" the connections in the model.
- Dependencies between signals on either side of an integrator block do not show up in the dependency graph. However, the signal downstream depends on the signal on the integrator's IC port.

`strongComp` takes as an input the sorted dependency graph obtained through `topSort` and returns a list of lists, each sublist a set of strongly connected components. Iff the length of every sublists is 1, then there are no strongly coupled components.

- To find strongly-connected components, we process the connectors in reverse *topological* order. In other words, the connections are visited in the same direction as in the model (as opposed to `topSort`).
- Once again, the dependencies between signals on either side of an integrator block do not show up here. However, the signal on an integrator's IC port depends on the signal downstream (symmetric to `topSort`).

The Time-Slicing simulator in implemented in the file `SIM_startResume.py+`, in the class `TS_Simulator` (replacing the class `Generator` of the original code). Note that this class is implemented twice: see below for explanations (only look at the first implementation for now).

The algorithm follows closely the pseudo-code. To support the `PAUSE` and `RESUME` features of the AToM3 environment, the simulator itself is split into two methods:

`initSimul` Initialize the time-slicing simulator: Topological sorting, and detection of algebraic loops.

`mainLoop` Main simulation loop

The other methods *added* to class `TS_Simulator` are:

`processDelayBlocks` Process all delay blocks (integrators). This is the Euler-Cauchy approximation:

$$x_1 = x_0 + h \cdot f(x_0)$$

with:

- x_0 – the integrator's current state (initial `block_out_value` value)
- x_1 – the integrator's next state (updated `block_out_value` value)
- h – the time step (`model.simul_delta_t.getValue()`)
- $f(x_0)$ – the current slope (`block_tmp_value` value)

`processConnectors` Process all connectors in `topsort` order. The integrator blocks' IC port is used only at initialisation, i.e. when `self.currTime == self.initTime`.

`sendOutput` Send outputs to plotter. A wrapper around `TS_Simulator.sendPlotter`, itself a wrapper around `model.dataSet.append`.

We note that algebraic blocks must be processed in different ways:

- ***n*-ary blocks:** some blocks (`Adder`, `Product`) implement *n*-ary operations. These blocks need to be reset before every simulation step. Resetting is done in `processConnectors` by placing the identity operator for the supported operation (0.0 for addition, 1.0 for multiplication) in the blocks' `block_out_value` attribute.
- **Generic blocks:** `Generic` blocks are unary functions. The function implemented is stored as a string in the block's `block_operator` attribute (e.g., `'sin'`, `'cos'`...). For now, only those functions where `fnc(x)` is meaningful in Python are supported (where `block_operator == 'fnc'`, and `x` is a float).
- **Clocks:** `Time` blocks can be treated in different ways. Our decision was: whenever a connector tries to read a clock's `block_out_value`, the clock gives its initial value plus the elapsed time (i.e., the current time - the initial time).

The control of the simulation from the AToM3 environment, through the `START/RESUME`, `PAUSE` and `RESET` buttons, works OK. For this, we needed to modify class `SIM_startResume.SimulatorThread` (replacing the class `GeneratorThread` of the original code), and the event function `SIM_startResume.startResume`. Files `SIM_reset.py` and `SIM_pause.py` needed to be modified as well, but `MODEL_plotWindow.py` was left untouched.

It is possible that some AToM3 operations result in unexpected behaviours (e.g., switching models or modifying a model during a session, leaving AToM3 during a simulation, playing too much with the plot window...)

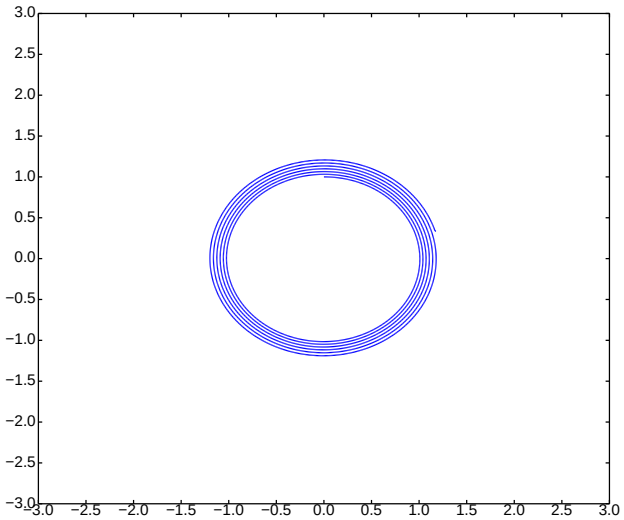

Figure 6: Circle Test with $\Delta t = 0.01$ (x vs y)

For the circle test problem, the analytical solution should look as a circle in phase space. However, since we use a simple Euler-Cauchy (first-order) approximation we expect that our numerical solution in phase space will look more like a spiral. The bigger the time-step, the more dramatic the effect. The following figures confirm this (compare with Figure , which was obtained with Runge-Kutta-Fehlberg — fourth order with embedded fifth order):

To find the optimal solution for the ballistic problem, we hacked the file `SIM_startResume.py`: the second definition of class `TS_Simulator` is solely dedicated to solve this problem (we actually pasted and modified the whole original class — this is not elegant, but faster...). It does several simulations of the ballistic problem, with the initial angle varying between 0.0° and 90.0°. The simulation now terminates when the height of the projectile reaches 1. After each simulation we plot, with respect to the angle, both the length of the simulation and the absolute difference between the projectile distance and its target distance (i.e., 30.0).

Results are shown below: the horizontal axis is the angle (in degrees). In blue, the duration of the simulation (multiplied by 6). In red, the absolute difference between projectile distance and target distance. We see the target is reached when the angle is approximately 22°, and 67°. But the time to reach the target is much smaller in the first case. This is the solution we were looking for!

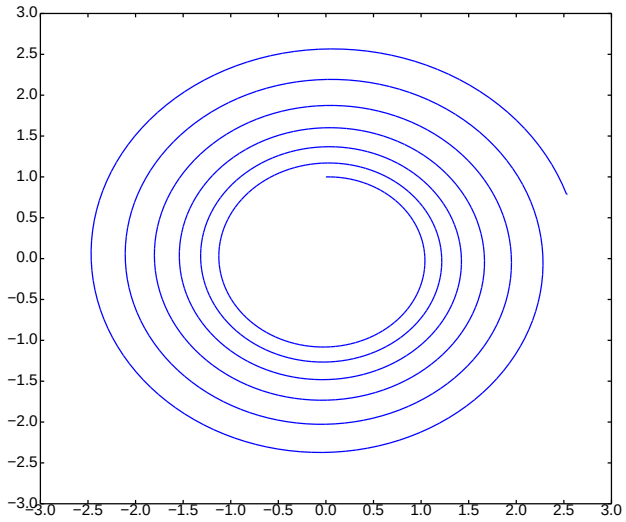

Figure 7: Circle Test with $\Delta t = 0.05$ (x vs y)

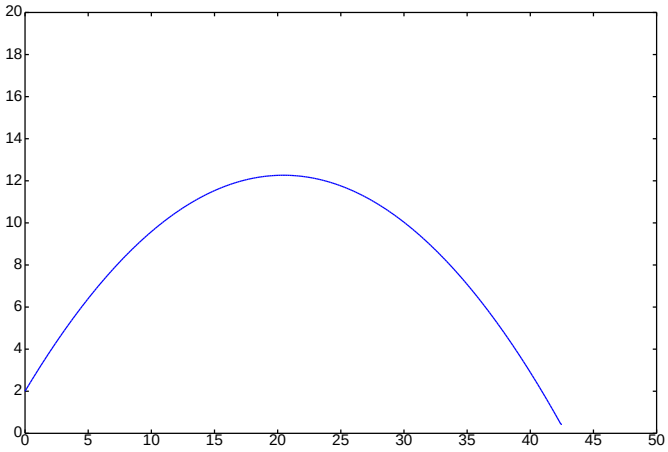

Figure 8: Ballistic Problem (x vs y, $\theta = 45°$)

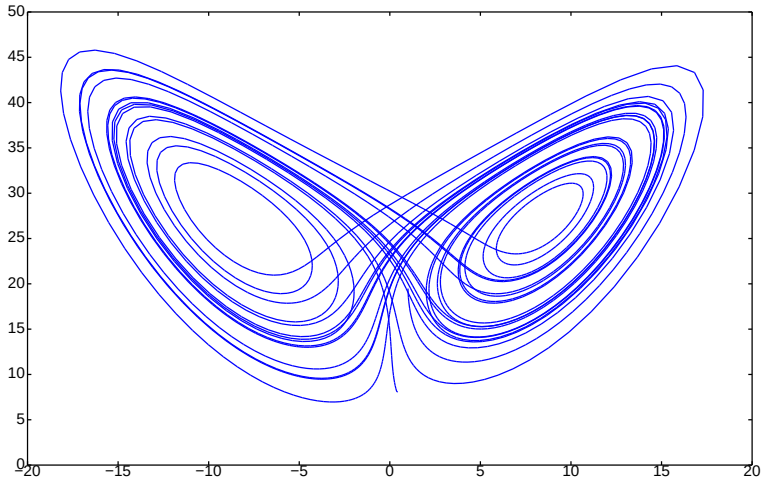

Figure 9: Lorenz Attractor (x vs z)

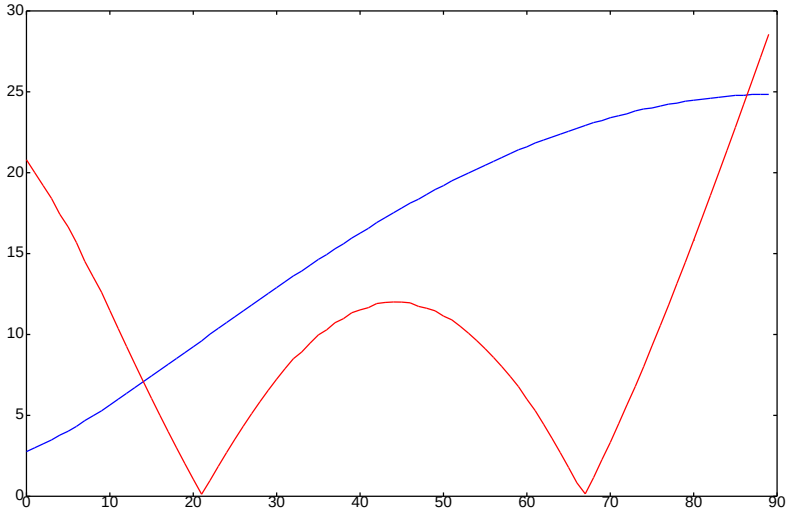

Figure 10: Result for Study Case

The analytical solution for the ballistic problem can be found to be:

$$\theta = \arccos\left(\frac{15\sqrt{2}g}{v\cdot\sqrt{g+v^2\pm\sqrt{v^4+2gv^2-900g^2}}}\right)$$

$$t = \frac{\sqrt{2}}{g}\sqrt{g+v^2\pm\sqrt{v^4+2gv^2-900g^2}},$$

where v is the speed, and g is the gravitational acceleration. Hence with $g = 9.81$ and $v = 20$, we get $\theta \approx (66.75°, 21.34°)$, and $t \approx$ (3.80s, 1.61s). This is consistent with our experimental results.