# Turing Centenary Competition Entry

## Overview of the Program

Alongside the Javadoc (which can be found in the project folder) and source code, this document aims to explain how our program functions.

This software is a Turing Machine Simulator which takes the contents of the tape and the instruction set as inputs and then visually displays a simulation of the given Turing Machine. The software is able to simulate Turing Machines in both one and two dimensions. *(As a result, the additional directions up and down have been added to the valid directions for a quintuplet – U and D respectively).*

First of all, it is worth noting that the simulator makes use of Java interfaces. This allowed us to develop the GUI and simulator separately. A rudimentary simulator was used to test the GUI in development.

During testing, some of the test data was found to be incorrect. We were unsure if this was intentional and so corrected test data has been included (in the test data format) with the software.

## Turing Machine Simulator

The simulator has the following:

- List of instructions/quintuplets, as a `List<Instruction>`.
- The tape, as a `Tape`.
- The current state, which is initially assumed to be 0 unless the user changes it via the GUI.

To execute an instruction, the simulator must first find the instruction for the current symbol under the tape head and the current state. To do this, it calls the `findInstruction(int currentState, char inputSymbol)` method.

The simulator already knows the current state, but it needs to get the symbol under the tape head from the Tape object. It does this by using the Tape's `getTapeSymbolAt(int x, int y)` method.

It then iterates over the list of `Instructions` until it finds an `Instruction` with the correct input symbol and input state. If more than one `Instruction` is found, it throws an exception, as there should only be one `Instruction` with a specific input symbol and state pair in the instruction set.

Once it has found the correct `Instruction`, it attempts to execute the instruction by calling the `executeInstruction(Instruction)` method.

This method first requests the current x/y co-ordinates of the tape head from the `Tape`. It then uses the `Tape`'s `setTapeSymbolAt(char symbol, int x, int y)` method to write `instruction`'s output symbol to the `Tape`.

The instruction then uses a switch statement to move the tape head in the correct direction based on its `direction` field. Constants were used for the direction (`Instruction.MOVE_LEFT`, `Instruction.MOVE_RIGHT ... Instruction.HALT`) to improve code readability.

This method then adds the executed `Instruction` to the history of instructions and updates the GUI.

## Executing Step-by-Step

The above description outlines how a single instruction is executed using the `step()` method from the `TuringSimulator` class. Each time the user clicks the step button in the GUI, the `step()` method is executed.

## Continuous Playback

To play the simulation continuously, the simulator calls the `step()` method inside a nested while loop. The paused flag represents whether the simulation is currently playing.

To have the appearance of playing the simulation at a steady rate, after executing an instruction execution of the simulator thread is paused for a certain number of milliseconds. This does not affect the GUI as it runs in a separate thread.

### Adjusting the Speed of Simulation

The `speed` variable can be modified by a `JSlider` in the GUI, allowing the user to speed up or slow down the simulation while the simulation is playing. See the GUI section for more information.

## Pseudo Code

The main loop of the program can be described by the following pseudo code:

```
INT sleep;
BOOLEAN paused;

WHILE TRUE
    WHILE NOT paused
        CALL step()
        WAIT sleep MILLISECONDS
    END
END
```

The `step()` method can be described by the following pseudo code:

```
INT currentState;

INT x ← GET TAPE HEAD X;
INT y ← GET TAPE HEAD Y;

CHAR inputSymbol ← GET SYMBOL AT (x,y)

INSTRUCTION instruction ← FIND INSTRUCTION WITH INPUT SYMBOL inputSymbol AND INPUT
STATE currentState;

EXECUTE instruction;
```

---

The `findInstruction(int currentState, char inputSymbol)` method can be described by the following pseudo code:

```
INT currentState;
CHAR inputSymbol;

LIST<Instruction> instructionSet;

Instruction instruction;

FOR EACH Instruction IN instructionSet
    IF INPUT SYMBOL IS inputSymbol && INPUT STATE == currentState
        IF instruction IS NULL
            instruction ← INSTRUCTION;
        ELSE
            PRINT "Error: more than one instruction found!"
        END
```

```
    END
END

RETURN instruction;
```

---

The `executeInstruction(Instruction)` method can be described by the following pseudo code:

```
Instruction instruction;

INT x ← GET TAPE HEAD X;
INT y ← GET TAPE HEAD Y;

SET SYMBOL ON TAPE AT (x,y) TO OUTPUT SYMBOL OF instruction;

INT direction ← DIRECTION OF instruction;

IF direction IS LEFT
   TAPE HEAD X --;
ELSE IF direction IS RIGHT
   TAPE HEAD X ++;
ELSE IF direction IS UP
   TAPE HEAD Y --;
ELSE IF direction IS DOWN
   TAPE HEAD Y ++;
ELSE IF direction IS HALT
   PAUSE SIMULATION;
   SET CURRENT STATE TO 0;
END
```

## Tape Data Structure

To hold the contents of the tape a new class was created: the `Tape` class. This class uses a two-dimensional `ArrayList` – an `ArrayList<List>` containing multiple `ArrayList<Character>`s - to store the contents of the tape. As the tape can be of infinite size, but is of finite size at any particular time, the `Tape` must be able to re-size its data structure dynamically.

To achieve this, we use the `setTapeSymbolAt(char symbol, int x, int y)` method to make changes to the symbols on the tape. This method detects if the cell at (x,y) is in the data structure, and if it is not, the tape is resized to accommodate it by inserting new rows or columns in the 2D `ArrayList`.

Initially the tape data structure consists of a single cell (0, 0). If the tape head is moved or a symbol is changed in a cell not in the data structure (initially any cell which is not [0, 0]) the data structure is resized. For example, say I edit the cell at (-4, -4) – A tape consisting of 1 column and 1 row would need to insert 4 rows and 4 columns at the beginning of the data structure. If I now edit the cell at (-1,-3) no rows or columns would need to be inserted as (-1, -3) is already in the data structure.

Note that when the size of the tape changes, because rows and columns may be inserted at the beginning of the 2D `ArrayList` the cell at (0, 0) on the tape may not be (0, 0) in the 2D `ArrayList` – i.e. `list.get(0).get(0)`. To counter this, the variables `originX` and `originY` are updated each time a change is made to the tape to reflect the new position of the cell (0, 0) in the 2D `ArrayList`. As in the above example, if I edit the cell at (-4, -4) `originX` would increase to 4 and `originY` would also increase to 4.

If I try to get a symbol in the cell at (x, y) on the tape the program would do `list.get(originY+y).get(originX+x)`. If that cell does not exist in the data structure a blank symbol (_) is returned.

## Saving and Loading Files

In addition to simulating a Turing Machine the software is able to load or save Turing Machines from or to a file.

The simulator can:

- Import files from the test data format.
- Open files in a custom format, the Turing Simulator File (.tsf).
- Save files in the Turing Simulator File (.tsf) format.
- Create a new blank simulation.

Before opening or importing files, the state of the simulator is reset to a blank simulation.

### Importing Files from Test Data

As the files used by our software differ from the format given in the test data, it was necessary to add a method to import files from the test data format. Given the specification for the test data format, we created a method which does the following:

The first line of the file is read to determine the number of instructions.

A for-loop is used to iterate over that number of lines. At each line, a new `TuringInstruction` is created from the quintuplet on that line and added to the instruction set.

After that many lines have been read, the simulator reads each subsequent line and adds it to a String variable. This results in one long string containing the entire contents of the tape.

The simulator then iterates over the string until it finds a '!' symbol, at which point it begins parsing the string as the contents of the tape. When it reaches a '<' symbol, it checks whether there is a corresponding '>' symbol indicating the position of the tape head. If there is, it sets the position of the tape head to the symbol between '<' and '>', adds that symbol to the tape, and jumps ahead to the '>' symbol to continue parsing. The position of the tape head will only be set at the first pair of '<'/'>' symbols. Any subsequent '<'/'>' symbols will be interpreted as part of the tape contents.

### Custom File Format

With the addition of a second dimension to the tape it was necessary to create a new file format. This format, the Turing Simulator File (.tsf), is defined as follows:

The first line is the number of instructions (quintuplets) in the file.

This is followed by each quintuplet on a separate line.

The line after the instruction set is the dimension of the tape – either 1 or 2.

The following lines are the contents of the tape. Each row of the tape is on a separate line. The contents of each row of the tape are preceded by a single '!' symbol. On one row, a number is placed before the '!' – this indicates that the position of the tape head is on this row at the position corresponding to the number before the '!'.

As an example, here is an implementation of Langton's Ant initially, then after 50 steps in the .tsf format:

*Initial*

```
8
(0,_,1,#,R)
(0,#,3,_,L)
(1,_,2,#,D)
(1,#,0,_,U)
(2,_,3,#,L)
(2,#,1,_,R)
(3,_,0,#,U)
(3,#,2,_,D)
2
!_______
!_______
!_______
!_______
5!_______
```

*After 50 Steps*

```
8
(0,_,1,#,R)
(0,#,3,_,L)
(1,_,2,#,D)
(1,#,0,_,U)
(2,_,3,#,L)
(2,#,1,_,R)
(3,_,0,#,U)
(3,#,2,_,D)
2
!________
!________
!________
!____##_
!___#__#
6!__#___#
!__#__#_
!___#__#
!____##_
```

These files are included with the software as *langton_ant.tsf* and *langton_50.tsf*, along with the state of the Turing Machine after over 10,000 steps when the ant has generated a highway as *langton_highway.tsf.*

This implementation of Langton's Ant shows three things about the program:

- The software is able to implement two-dimensional Turing Machines.
- The data structure we created to represent the tape resizes dynamically.
- The software can run continuously for many thousands of steps without error.

## GUI Elements

To begin, this document will cover the features requested in the guidelines for the competition, namely:

- Display the current state
- Display the current instruction
- Display the cell under the tape head
- Display a section of the tape around the tape head

Additionally, the GUI can:

- Display one-dimensional and two-dimensional tapes.
- Display any region (of a specific size) on the infinitely long tape.
- Allow the user to edit the tape from within the program.
    - The tape may be edited while the simulation is in progress.
- Display the co-ordinates of the tape head on the tape relative to an origin cell.
    - Currently this information is not saved, so it is to be used for reference only and you should not rely on it being consistent between sessions.

- Follow the tape head as it moves across the tape.
- Navigate across the tape, showing a section of the full tape.
- Display a history of previously executed instructions.
    - Up to 50 are held in the history but the number displayed is limited by screen resolution – a minimum of 8 on a 1680x1050 display at full screen.
- Display the current instruction set.
- Allow the user to edit the instruction set from within the program.
- Allow the user to pause, resume and step the simulation.
    - The speed of the simulation may be adjusted while the simulation is on progress.
- Direct the user to a help page located at http://ayrtonmassey.github.com/Turing-Simulator/
    - This works in Windows operating systems only. Other operating systems will receive the following error:

      "It looks like you're using *OPERATING SYSTEM NAME*. This software can only open the help page itself in Windows. Please point your browser at: http://ayrtonmassey.github.com/Turing-Simulator/"

## Displaying the Tape

The view of the tape in the GUI consists of a `JPanel` containing:

- a `JTable` with a custom cell editor and table model displaying a section of the `Tape`.
- the Tape Head, which is drawn to the `JPanel` using the methods from the `java.awt.Graphics` class.

### Tape Contents

The tape contents are displayed in a `JTable`. The coordinates of the top left and bottom right cell on the section of the tape which is displayed are stored. Each time the GUI is updated with the `update()` method, the `JTable` is updated with the contents of the `Tape` between (`tapeBeginRowIndex`, `tapeBeginColumnIndex`) and (`tapeEndRowIndex`, `tapeEndColumnIndex`).

### Tape Head

The tape head is drawn to the screen using the `java.awt.Graphics` methods `drawString`, `fillRect` and `fillPolygon`. The tape head is drawn in three parts: The arrow head as a polygon, the main body of the tape head as a rectangle, and the current state using the `drawString` method. To enable subscript text two `Fonts` were defined: One of regular size and the other a smaller size for subscript text.

Java does not appear to support subscript text with `java.awt.Graphics` so calculations are performed to find the correct x/y position of the text and subscript text and width of the tape head before it is drawn to ensure everything is positioned correctly.

### Editing the Tape

The `JTable` which displays a section of the tape has a custom cell editor, the `TapeCellEditor`. This class supplies the cell editor component (the text field displayed when a cell is being edited). Before input is accepted and added to the tape validation is performed to ensure that only a single character has been entered in to the text field (or nothing, in which case a blank is inserted). This validation is simply a format check on the contents of the text field:

```
String input;
IF LENGTH OF input <= 1 //symbol or blank
    IF LENGTH OF input == 1
        ADD input TO TAPE //symbol entered
    ELSE
        ADD '_' TO TAPE //nothing entered so cell is blank
    END
END
```

### Displaying Tape Head Coordinates

The coordinates of the tape head are stored in the Tape instance. When the GUI is updated using the `update()` method, the coordinates of the tape head are retrieved from the Tape and displayed in a `JLabel` in the GUI.

### Centring the View on a Cell

To centre the view on a particular cell, the tape display calculates the number of rows and columns before and after that particular cell which will be displayed in the table – the number of rows or columns in the table divided by 2 and then rounded down (the number of rows and columns should be an odd number). This number is used to calculate `tapeBeginRowIndex`, `tapeBeginColumnIndex`, `tapeEndRowIndex` and `tapeEndColumnIndex`. When the GUI is updated the view of the tape will be centred on the given cell.

### Following the Tape Head

Each time the GUI is updated, if the `followTapeHead` flag is set to true then the view of the tape is centred on the tape head's position – (`tape.getTapeHeadX()`, `tape.getTapeHeadY()`) where tape is an instance of `Tape`. If the flag is false, no change is made to the tape view.

When the `JButton` in the GUI is clicked `followTapeHead` will be set to true if it is false or vice versa.

### Navigating the Tape

There are 4 `JButtons` in the GUI for navigating the tape, one for each direction: up, down, left, right. When one of these buttons is clicked, `tapeBeginColumnIndex` and `tapeEndColumnIndex` are increased or decreased by 1 to move the tape viewport left or right, `tapeBeginRowIndex` and `tapeEndRowIndex` are increased or decreased by 1 to move the tape viewport up or down.

## Displaying Instructions

The `drawString` method of the `java.awt.Graphics` class is used to draw instructions to the screen. Instructions (or quintuplets) are displayed like this:


*(Enlarged)*

To enable the program to draw instructions in this way calculations are performed to find the correct x/y position of the text and subscript text before it is drawn to ensure everything is positioned correctly.

### Instruction Set

The instruction set is displayed as a `JList` contained within a `JPanel`. The `JList` uses a custom cell renderer to draw instructions so that subscript text is supported.

The `JPanel` also contains two `JButtons`: one to add an instruction to the instruction set and another to remove the selected instruction.

The add button creates a new `TuringInstruction` from a quintuplet (by reading the input state, symbol etc. from the quintuplet) and adds it to the `JList`'s list model. The simulator's instruction set is updated to reflect this change.

The remove button removes the selected `TuringInstruction` from the `JList`'s list model. The simulator's instruction set is updated to reflect this change.

Double-clicking an instruction allows you to edit it. When an item in the list is clicked, the program checks the number of clicks and if it is greater than or equal to 2, the edit dialog is opened. The quintuplet entered by the user generates a new `TuringInstruction` which replaces the old `TuringInstruction` in the `JList`. The simulator's instruction set is updated to reflect this change.

### Instruction History

Each time the simulator executes an instruction successfully, that instruction is added to the history of executed instructions (an `ArrayList<Instruction>`) kept by the GUI. The history of instructions is drawn to a `JPanel`. Each instruction in this list is drawn in reverse order from the bottom of the `JPanel` to the top to ensure the most recent instruction is shown in the correct place and previous instructions before it.

If the size of the instruction history goes above 50 the instruction at position 0 in the list is removed.

## Pausing and Resuming Simulation

As described earlier the main loop of the simulator is a nested while loop:

```
BOOLEAN paused;
WHILE TRUE
    WHILE NOT paused
        //execute instruction
    END
END
```

Instructions will only be executed if `paused` is false (i.e. the simulation is in progress).

To pause the simulation, a `JButton` in the GUI sets the `paused` flag to true and then disables the pause and step buttons. The play button is enabled.

To play the simulation, a `JButton` in the GUI sets the paused flag to false and then disables the play button. The pause and step buttons is enabled.

To advance the simulation by one step while paused, a `JButton` in the GUI calls the simulator's `step()` method.

### Controlling Speed

A `JSlider` in the GUI modifies the `sleep` variable in the simulator. The simulator sleeps for `sleep` milliseconds each time an instruction is executed in the main loop (not when using the `step()` method). This allows the user to control the delay between instructions being executed, and therefore the speed of simulation.

## File Management

A `JMenuBar` along with `JMenus` and `JMenuItems` provides access to the methods which open, save, import or create new files. Each time a file is created or imported the GUI must be reset to an initial state. The `reset()` method in the GUI does this by removing all its components and then calling the `init()` method to add new fresh components.