TUHH
Technische Universität Hamburg-Harburg

Institut
für
Regelungstechnik

User Manual

Andrey Popov

FormAtion Simulation Tool - FAST

February 2010 Report 2010.03

Contact data:
Andrey P. Popov
Institute of Control Systems
Hamburg University of Technology
Eissendorfer str. 40
21073 Hamburg
Germany

andrey.popov@tu-harburg.de
http:\\www.tu-harburg.de\rts

Contents

1 Introduction

The *FormAtion Simulation Tool* - FAST - is a tool for Matlab developed in [Ros09] for the purpose of performing realistic simulations of multi-agent systems (MAS). FAST allows simulating linear and non-linear discrete-time agents, that communicate with each other. The communication links can be additionally subject to time-delays, range restrictions and obstacles. FAST is developed and tested under Matlab 2009a.

1.1 Features

As by now, Feb 2010, FAST provides the following functionality.

- Simulation of MAS with discrete-time dynamics.
- Linear and non-linear agent dynamics (the non-linear will require manually specifying the dynamics).
- Adding/Removing agents during the simulation - requires pre-specifying the time of change and the agents to be added/removed.
- Changing the communication topology during the simulation - requires pre-specifying the time of change and the new topology.
- Defining communication obstacles with ball or parallelogram shape.
- Fixed communication delays with or without delayed own outputs of the agents.
- Specifying communication range, i.e., distance dependent communication.
- Graphical User Interface (GUI) for easy control and visualization.
- Export of the simulation data.

1.2 License

1.3 Download and Development

FAST can be downloaded by the development repository at
http://bitbucket.org/fcmas/fast/

The latest significant release can be directly downloaded as from the following link
http://bitbucket.org/fcmas/fast/downloads/FAST.zip

If you are interested in providing additional functionality to FAST, improving the code, or modifying the GUI you are welcomed to do so. Please send a short note to the authors, using the contact on the second page of this document, so that you can obtain *Writer* rights to the repository.

1.4 Installation

To install FAST add the main folder where you have decompressed the tool to the Matlab path

- Use `addpath('D:\mytoolboxes\FAST')` for a single use of the tool, where you need to change `D:\mytoolboxes\FAST` to the correct path on your computer.
- *File→Set Path ...* if you plan to use the tool often.

2 Toolbox Structure

The toolbox is created using the object-oriented capabilities of Matlab version 2008a and later. The main objects in the toolbox are:

- Agents - each object agent represents a physical agent and hence includes both a model of the agent and the local controller. Key properties of the agent are its states, and a key method is a function that computes the outputs and the states for the next time-step.
- Formation - this agent comprises all agents and serves as a mediator between the simulation environment and the agents. This object takes care of updating the communication topology when agents get out of range, or an obstacle interrupts a communication channel.
- Simulation environment - that is an object that takes care of the progress of the simulation (i.e., main simulation loop) and performing any necessary animations.
- Animator - objects used with the graphical user interface (GUI) of the toolbox to allow easier control of the simulation and visualization of the data.

For more information on programs, please refer to [Ros09].

3 Graphical User Interface

The easiest way to perform a simulation of a multi-agent system is by using the graphical user interface of FAST. It is possible to perform a simulation of a MAS from a command line in Matlab, but is not discussed here, as it requires manually constructing all objects (Agents, Formation, Simulation Manager, etc.) in the correct order.

To start the GUI of FAST just evoke `FAST` in Matlab. A snap-shot of the GUI is shown in Figure 1.

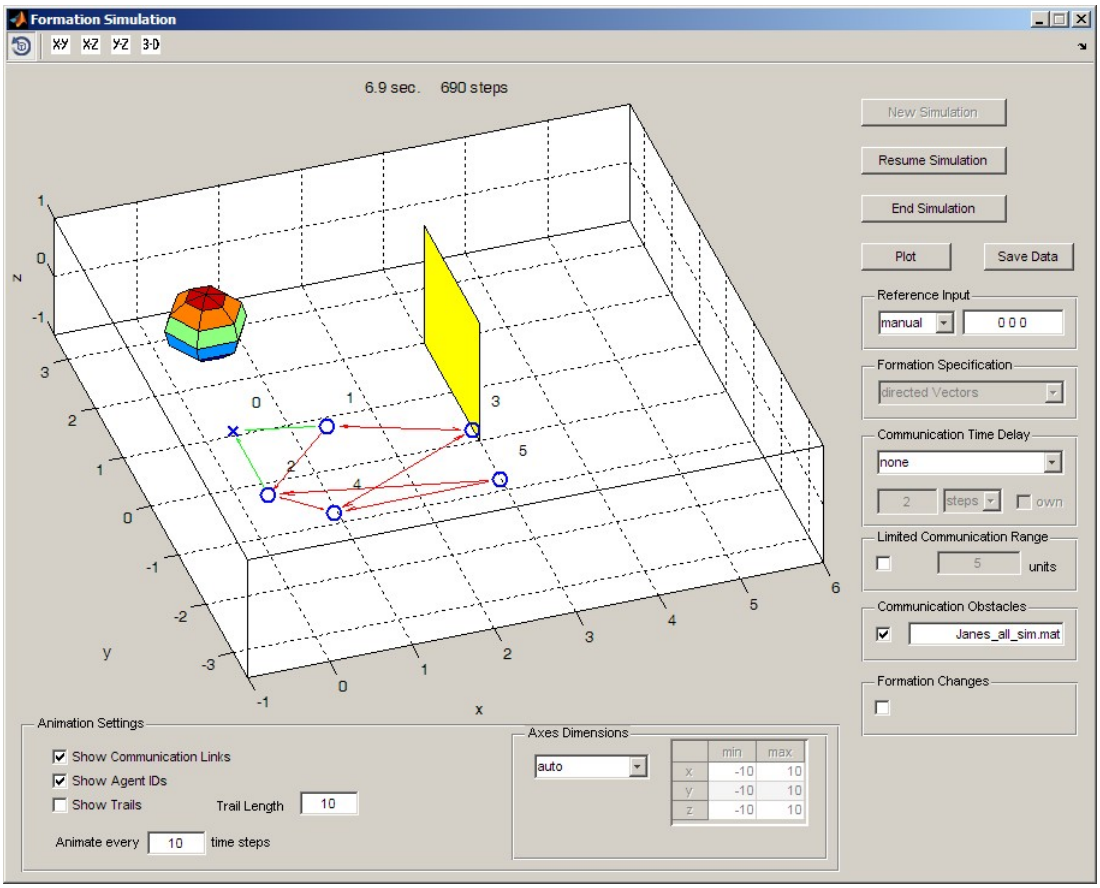

Figure 1: Snap-shot of the GUI of FAST

The GUI is split on 3 main parts. The big window in the upper-left part is where the formation is visualized. The buttons to the right of it are for controlling the simulation, and the switches under that window are for adjusting what and how is visualized. As most of the buttons and switches are self-explanatory they will not be covered here.

The style of the animation, i.e., with what symbols and colors are the agents, trails and arrows visualized can be adjusted manually via the settings structure in the object *Animator*.

4 Tutorial on Defining and Simulating Formations

This tutorial is intended for new users of FAST and aims at giving an overview of the main steps in defining and performing a simulation of a multi-agent system. The tutorial is written in a story-like fashion in the hope that this will make it easier to read. The code to this tutorial can be found subfolder `\demo` of FAST.

4.1 A simple simulation

Our story starts with Jane, who is a researcher in the area of cooperative control of agents. One day Jane discovers a rather interesting example of a formation flight of quad-rotor helicopters (quadrocopters) and decides to perform some simulations. While reading the details, she is at first taken aback when she learns that the quadrocopters have non-linear and unstable dynamics. Luckily, she discovers a linearized, discrete-time model $P(z)$ with 12 states, 4 inputs and 15 outputs, and a corresponding controller $K(z)$, and decides to set up a simple simulation in FAST.

As Jane understands it from [PPW09] the closed-loop of agent and controller has the form, as shown in Figure 2 – the first 3 outputs of each quadrocopters (y_i) are simply the coordinates of the agent, and the next 12 outputs (ϕ_i) are the states. The controller uses the state information to locally stabilize the quad-rotor, and the other 3 outputs are send to other agents.

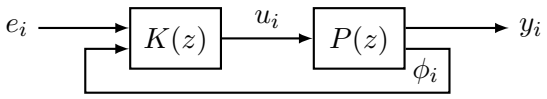

Figure 2: Local feedback of single agent and its controller

Jane is further delighted to understand, that the additional input e_i of the controller is the average error of that particular agent in the multi-agent system, which corresponds to what she has been using previously:

$$e_i = (r_i - y_i) - \frac{1}{\mathbf{J}_i} \sum_{k \in \mathbf{J}_i} (r_k - y_k),$$

where $\mathbf{J}_i$ is the set of agents from which agent i receives information, and r_i are commanded positions. At this place, Jane makes a mental note, that if she doesn't include a leader or an absolute reference to the formation later, the whole multi-agent system will attain the "shape" specified by r_i, but not necessary on the positions r_i.

Having understood all that Jane sets upon defining a formation of $N = 5$ agents in FAST. First she loads the system model and the controller, and defines the dynamics of the agents. As she is not a big fan of loops, she decides to do it by hand for now, so that it is easier to make simple changes. She sets the initial states of all quad-rotors to zero, and positions them on a distance of 1 unit from each other on the x-axis.

```
% 1. Load the agent and controller models
load copter_models P K  % plant P and controller K

% 2. Specify the agents        init. state  init. pos.
models = [];                                % x y z  - coordinate
models{1} = LinearModel( P, K, zeros(12,1), [1 0 0]' );
models{2} = LinearModel( P, K, zeros(12,1), [2 0 0]' );
models{3} = LinearModel( P, K, zeros(12,1), [3 0 0]' );
models{4} = LinearModel( P, K, zeros(12,1), [4 0 0]' );
models{5} = LinearModel( P, K, zeros(12,1), [5 0 0]' );
```

Jane knows, that by specifying the initial position as a 3 dimensional vector, she is actually specifying the dimension of y_i, from which the function `LinearModel` automatically deduces the dimension of ϕ_i to be the number of outputs of $P(z)$ minus 3, i.e., 12.

Next, Jane specifies which of the signals in y_i should be plotted along which coordinate. As she wants to obtain a 3-dimensional plot and as the signals in y_i are already in the correct sequence (x-y-z), she just defines

```
% 3. Spatial Coordinates
output2Coordinates = [
    1 0 0;
    0 1 0;
    0 0 1];
```

Had she wanted to make a 2-d visualization in the x-y plane, she could have defined it as follows.

```
% 3. Spatial Coordinates
output2Coordinates = [
    1 0 0;
    0 1 0];
```

In both cases, Jane knows, that all three elements of y_i will be saved from FAST for later analysis.

In the next step Jane sets upon defining how the agents communicate with one another. Late she will want to try some changing topologies, but at first she decides to keep it simple. Keeping in mind, that the adjacency matrix has to be $N + 1 \times N + 1$, since for FAST agent with number 0 specifies a direct reference input (virtual agent without dynamics), she defines the topology in Figure 3 by the following 6×6 matrix.

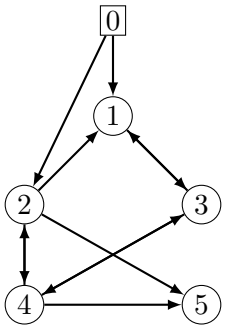

Figure 3: Graph representation of the communication topology with $N = 5$ agents and reference input (0).

```
% 4. Adjacency matrix
%    agent # 0 1 2 3 4 5
adjacency = [0 0 0 0 0 0; % this is the Reference point
             1 0 1 1 0 0; % agent 1: receives from ref. point, agents 2 & 3
             1 0 0 0 1 0; % agent 2: receives from ref. point, agent 4
             0 1 0 0 1 0; % agent 3: receives from agents 1 & 4
             0 0 1 1 0 0; % agent 4: receives from agents 2 & 3
             0 0 1 0 1 0];% agent 5: receives from agents 2 & 4
```

Finally, Jane defines the reference inputs to the 5 agents, compared to the virtual reference (*agent 0*). For now, Jane does not want to define a reference sequence for the virtual reference point, but will rather use the manual control available in the GUI of FAST. She knows that there will be some more information about that subject in Section 4.5.

```
% 5. Absolute reference input for all agents:
absRefInput = [
     0  -1  0;    % agent 1
    -1  -2  0;    % agent 2
     1  -2  0;
    -1  -3  0;
     1  -3  0;];  % agent 5
```

Finally, she saves all the necessary data in a MAT file named, `Janes_init.mat`. Jane knows what actually matters is not the name of the MAT file, but rather the name of the variables. She always pays attention when saving her data, so that she avoids errors and problems later on.

```
save Janes_init.mat   models output2Coordinates adjacency absRefInput;
```

Next she evokes `FAST`, clicks on **New Simulation** and selects the file `Janes_init.mat`. From now on it is all quite easy. Jane starts the simulation by **Start Simulation** and lets it run for several seconds. Then, because she doesn't want to see the fine details, but the overall behavior

she types 20 in the **Animate every** [ ] **time steps** box, which forces FAST to visualize the position only after every 20 steps. Further, Jane *un-checks* the **Show trails** box in the bottom right corner and uses the small button in the upper-left corner to rotate the view and see the formation from different points. At about 19th second she pauses the simulation with **Interrupt Simulation**, and defines [0 2 1] as a new reference point for the formation, by typing the tree numbers in the **Reference Input** box. After some more time Jane decides that she has seen enough for the time and ends the simulation. Because she is not yet sure, whether she will use the data later, she decides to saves them by clicking on **Save Data** and typing in an appropriate name.

4.2 Communication Delays

Having succeeded with her first simulation of a quad-rotor formation flight Jane decides to explore the effect of communication delays. For the purpose she selects **New Simulation** and loads `Janes_init.mat` again. Next, she selects the drop-down box under **Communication Time Delay** on the right side of the GUI and is somewhat disappointed to find out that the only two options are **none** and **fixed and equal**. She writes to the authors of FAST and asks why no distance dependent or different delays are supported. By the reply she learns that actually on their to-do list.

As Jane knows, when the **own** check-box is marked the own outputs of the agents are also delayed when computing the error e_i. So she sets upon trying different time-delays both with own and without own delay, and discovers that whereas even several second delay can make the formation with own-delays *unstable*, this was not possible in the absence of own-delays... even for a big ones. Somewhat puzzled she contacts the people who developed the controller she is using, who inform her that this is a special feature of their controller and is due to a small-gain condition they have employed [PW09]. This makes Jane wonder if she could use that same approach for her own simulations later on.

4.3 Communication Range

Jane decides to make the agents communicate only if the distance between them is shorter than 4 units, and easily achieves this by ticking the check-box by **Limited Communication Range** and entering `4` in the box. But then she faces a new problem – she cannot give a reference point far away, since than the communication brakes down and agents 1 and 2 do not receive the reference signal. She discusses this with a colleague of hers, who has more experience with FAST, who tells her, that there is a simple solution to this: she simply needs to define the entries in the adjacency matrix that shouldn't be distance dependent with a negative sign. Jane gives it a thought an decides that only the communications between the reference signal and agents 1 and 2 need to be unbreakable for her purposes, so she redefines the adjacency matrix as follows.

```
% 4. Adjacency matrix
%    agent #  0 1 2 3 4 5
adjacency = [ 0 0 0 0 0 0; % this is the Reference point
             -1 0 1 1 0 0; % agent 1: receives from ref. point, agents 2 & 3
             -1 0 0 0 1 0; % agent 2: receives from ref. point, agent 4
              0 1 0 0 1 0; % agent 3: receives from agents 1 & 4
              0 0 1 1 0 0; % agent 4: receives from agents 2 & 3
              0 0 1 0 1 0];% agent 5: receives from agents 2 & 4
```

and does not forget to export the whole settings as a MAT file.

```
save Janes_init.mat   models output2Coordinates adjacency absRefInput;
```

4.4 Communication Obstacles

Already having some experience with FAST, Jane decides to try and add some communication obstacles. As she had learned from [Ros09], in FAST the agents do not collide (yet), but the communication connections can be broken due to obstacles. Jane decides to try out both of the currently supported obstacles by FAST - balls and walls. She defines them and saves them to a file named `Janes_obstacles.mat`. The ball obstacle is intuitive to define, and as Jane is not primarily interested in great graphics she sets the number of faces visualizing the ball obstacle to 6. The wall obstacle gives her a bit of a trouble at first, until she realizes that by wall is meant just a parallelogram, defined by one corner and two offsets, i.e., with corners: `corner`, `corner+d1`, `corner+d1+d2` and `corner+d2`.

```
% Ball-shaped obstacle, where 'faces' is used for visualization
obstacles{1} = struct('type','ball','center',[0 1 1],'radius',0.5,'faces',6);
% Wall-shaped obstacle
obstacles{2} = struct('type','wall','corner',[4 0 -1],...
    'd1',[0 0 2],'d2',[0 1 0]);

% Save the obstacles to a MAT file
save Janes_obstacles.mat obstacles
```

Then, after starting `FAST`, she ticks the check-box by **Communication obstacles** and selects `Janes_obstacles.mat`, after which she starts the simulation. Later she understands that the advantage of specifying the multi-agent system and the obstacles separately is that one can load a new set of obstacles during the simulation.

Using the manual **Reference Input** Jane tries driving the formation through and around the obstacles and finds that it works as expected. Her only complaint is that the whole simulation seems to take longer, but, as she later learns, that is due to the fact that FAST needs to check for intersection of each communication link with each obstacle, and there seems not to be much one could do to avoid the slow-down.

4.5 Reference Input from File

During the last simulations it was tiring for Jane to adjust the reference signal to the *virtual reference agent* manually, so she decides to define a reference trajectory through a file. She decides to make the formation move with a constant speed in y-direction and perform sinusoidal oscillations in x-direction, while keeping the z-axis (the vertical one) at 1 unit. This she defines through the following code and saves the reference signal in `Janes_reference.mat`, where again the important part is the name of the variable `formationRefInput`.

```
t = 0:1000;            % time-steps
x = 2 * sin(0.01*t);   % x-axis
y = 0.002 * t;         % y-movement with speed of 0.002 units per time-step
z = ones(1,length(t)); % constant z-position
formationRefInput.t_step = t;
formationRefInput.refPos = [x; y; z];

save Janes_reference.mat formationRefInput
```

If Jane has left `formationRefInput.t_step` empty, then the reference signal will considered as one starting from the current time moment.

Alternatively Jane could have defined only few key points, like that

```
%% Specify reference key points
formationRefInput.t_step = [0, 200, 500, 1000];    % ref. pos. at time steps
formationRefInput.refPos = [
    0 0 0;          % at 0
    0 2 0;          % at 200
    0 5 1;          % at 500
    0 6 2]';        % at 1000    % note the transpose!
```

To change the distance between the agents (and not only the position fo the reference point) Jane moves on to the next section.

4.6 Formation Changes

Communication Topology Change

One last thing that Jane wants to simulate is the behavior of the above quad-rotor formation, when the communication topology is changing. She decides that she wants to perform only two changes to the topology at time steps 100 and 200, correspondingly. At the first change she wants to remove the link from the reference to agent 2 and add a link from agent 1 to agent 2. At the second change time, she wants to switch to complete communication topology, but leave agent 1 as the only one receiving information from the leader. All this, she defines via the following `changeData` cell-array.

```
changeData = [];
changeData{1} = struct(...
    'type',       'commTopology',...
    't_change',   100,  ...   % change occurs at this time step
    'adjacency',  [
     0 0 0 0 0 0;
    -1 0 1 1 0 0;
     0 1 0 0 1 0;
     0 1 0 0 1 0;
     0 0 1 1 0 0;
     0 0 1 0 1 0]);

changeData{2} = struct(...
    'type',       'commTopology',...
    't_change',   200,  ...   % step at which the change should occur
    'adjacency',  [
     0 0 0 0 0 0;
    -1 0 1 1 1 1;
     0 1 0 1 1 1;
     0 1 1 0 1 1;
     0 1 1 1 0 1;
     0 1 1 1 1 0]);

 save Janes_changes.mat changeData
```

She starts `FAST`, marks the check-box under **Formation Changes** and chooses `Janes_changes.mat` from the menu and waits until steps 100 and 200 to see the desired changes in the topology.

There are 3 other possibilities for types of changes that Jane does not plan on using at the moment.

Formation Shape Change

For example the following code, changes the reference to the agents in the 250th time step, such that the agents should arrange themselves at a distance of 0.5 from each other along the y-axis and at an altitude of 1.

```
changeData{3} = struct(...
    'type',       'constellation',...
    't_change',   250, ...
    'AbsRefInput', [
         0  -1    1;    % agent 1  [x y z] - coordinates
         0  -1.5  1;    % agent 2
         0  -2    1;
         0  -2.5  1;
         0  -3    1]);
```

Adding Agents

If Jane wanted to add 2 further quad-rotor agents at time-step 300, the first one of which appears at coordinates $\begin{bmatrix}-5 & 0 & 0\end{bmatrix}$ and receives information from agent 1 and sends information to agents 3 and 4, and the second appears at coordinates $\begin{bmatrix}-6 & 0 & 0\end{bmatrix}$ one receives information only from agent 5, she should use a code like the following.

```
load copter_models P K  % plant P and controller K
new_agents = [];
new_agents{1} = LinearModel( P, K, zeros(12,1), [-5 0 0]' );
new_agents{2} = LinearModel( P, K, zeros(12,1), [-6 0 0]' );

changeData{4} = struct(...
    'type',            'addAgent',...
    't_change',        300, ...
    'newAbsRefInput',  [-2 -4 1; 2 -4 1],... % with respect to ref.point
    'adjacency', [
    0 0 0 0 0 0 0 0;
   -1 0 1 1 1 1 0 0;
    0 1 0 1 1 1 0 0;
    0 1 1 0 1 1 1 0;
    0 1 1 1 0 1 1 0;
    0 1 1 1 1 0 0 0;
    0 1 0 0 0 0 0 0;
    0 0 0 0 0 1 0 0], ...
    'models', {new_agents});   % note the {} brackets
```

Removing Agents

For example, to remove agents 2 and 3 in the 700th step time one would need the following code.

```
changeData{5} = struct(...
    'type',       'deleteAgent',...
    't_change',   700, ...
    'deleteIds',  [2 3]);
```

Quite happy with her simulations, Jane shows them to her colleague Marta. Marta is impressed when she sees the simulation, but is not quite happy with all those definitions and MAT files. Jane explains that one could write the definition of the group of agents and the changes more efficiently using loops or/and another toolbox, named Formation Analysis and Control Toolbox (FACT), that provides many useful functions. Marta agrees that this would be useful, but she actually would prefer to by able to make manual changes, so to say *on the fly* to the formation. After a lengthy discussion they both agree that the current functionality is useful, if one wants to define complicated scenarios that can be reproduced, but also some easy changes by hand would be nice. Since as they learn this is not on the to-do list of the authors of FAST, they think about contributing to the project themselves.

4.7 Non-linear agents

As Marta is herself interested in non-linear systems she wants to try out the original non-linear model of the quad-rotor. In general she would have needed to construct her own non-linear model, as a *child* of the object Model, using the object-oriented principle of inheritance. However this time she is lucky, as the authors of FAST have already implemented the non-linear model of the quad-rotor helicopters. This turns out to be really helpful weeks later, when she implements a non-linear model of a walking robot by using the non-linear model of the quad-rotor as a template. However for the moment all that she needs to do to change from the linear to the non-linear model is to replace the first code fragment with this one.

```
% 1. Load the agent and controller models
load copter_models K  % load only controller

% 2. Specify the agents           init. state  init. position
models(1) = QuadrocopterModel( K, zeros(12,1), [1 0 0]' );
models(2) = QuadrocopterModel( K, zeros(12,1), [2 0 0]' );
models(3) = QuadrocopterModel( K, zeros(12,1), [3 0 0]' );
models(4) = QuadrocopterModel( K, zeros(12,1), [4 0 0]' );
models(5) = QuadrocopterModel( K, zeros(12,1), [5 0 0]' );
```

4.8 A complete scenario

Together Marta and Jane shorten up the definition of the scenario Jane has been working on to the following one (can be found under `\demo\FAST_demo_complete.m` in the folder of FAST).

```
%% Specify Agents and References
% 1. Load the agent and controller models
load copter_models P K  % plant P and controller K

models = [];
% 2. Specify the agents        init. state  init. position
N = 5;  % num of agents
Init_Pos = [(1:N)', zeros(N,1), zeros(N,1)];
for ind = 1:2     % linear models
    models{ind} = LinearModel( P, K, zeros(12,1), Init_Pos(ind,:)' );
end
for ind = 3:N     % non-linear models
    models{ind} = QuadrocopterModel( K, zeros(12,1), Init_Pos(ind,:)' );
end

% 3. Spatial Coordinates
output2Coordinates = eye(3);    % x-y-z

% 4. Adjacency matrix
%    agent # 0 1 2 3 4 5
adjacency = [0 0 0 0 0 0; % this is the Reference point
            -1 0 1 1 0 0; % agent 1: receives from ref. point, agents 2 & 3
            -1 0 0 0 1 0; % agent 2: receives from ref. point, agent 4
             0 1 0 0 1 0; % agent 3: receives from agents 1 & 4
             0 0 1 1 0 0; % agent 4: receives from agents 2 & 3
             0 0 1 0 1 0];% agent 5: receives from agents 2 & 4

% 5. Absolute reference input for all agents:
absRefInput = [
    0   -1  0;    % agent 1
    -1  -2  0;    % agent 2
    1   -2  0;
    -1  -3  0;
    1   -3  0];   % agent 5


%% Specify obstacles
% Ball-shaped obstacle, where 'faces' is used for visualization
obstacles{1} = struct('type','ball','center',[0 1 1],'radius',0.5,'faces',6);
% Wall-shaped obstacle
obstacles{2} = struct('type','wall','corner',[3 0 -1],...
     'd1',[0 0 2],'d2',[0 2 0]);
```

```
%% Specify reference key points
formationRefInput.t_step = [0, 200, 500, 1000];    % ref. pos. at time steps
formationRefInput.refPos = [
    0 0 0;          % at 0
    0 2 0;          % at 200
    0 5 1;          % at 500
    0 6 2]';        % at 1000    % note the transpose!

%% Specify topology changes
changeData = [];

% Change topology
adj2 = adjacency;
adj2(1,3) = 0;      % remove link 0-->2
adj2(2,3) = 1;      % add link    1-->2
changeData{1} = struct(...
    'type',       'commTopology',...
    't_change',   100,  ...   % time step of change
    'adjacency',  adj2);

% Change topology
adj3 = adj2;
adj3(2:end,2:end) = ones(N); % make all communications
for i=2:N+1
    adj3(i,i) = 0;  % set the diagonal of the adj. to 0
end
changeData{2} = struct(...
    'type',       'commTopology',...
    't_change',   200,  ...   % step at which the change should occur
    'adjacency',  adj3);

% Chenge formation shape
changeData{3} = struct(...
    'type',       'constellation',...
    't_change',   250, ...
    'absRefInput', [
    0  -1    1;    % agent 1  [x y z] - coordinates
    0  -1.5  1;    % agent 2
    0  -2    1;
    0  -2.5  1;
    0  -3    1]);

% Add agents
load copter_models P K  % plant P and controller K
```

```
new_agents = [];
new_agents{1} = LinearModel( P, K, zeros(12,1), [-5 0 0]' );
new_agents{2} = LinearModel( P, K, zeros(12,1), [-6 0 0]' );

%                    agents    3  & 4   receive from the new agent
adj4 = [adj3, [0 0; 0 0; 0 0; 1 0; 1 0; 0 0];
    0 1 0 0 0 0 0 0;   % the first new agent receives form agent 1
    0 0 0 0 0 1 0 0];  % the second new agent recevies from agent 5
% # 0 1 2 3 4 5 6 7    - agent number
changeData{4} = struct(...
    'type',            'addAgent',...
    't_change',        300, ...
    'newAbsRefInput', [-2 -4 1; 2 -4 1],... % with respect to virtual ref.
    'adjacency', adj4, ...
    'models', {new_agents});                % note the {} brackets!

% Remove agents
changeData{5} = struct(...
    'type',       'deleteAgent',...
    't_change',   700, ...
    'deleteIds', [2 3]);

% Save everything to Janes_all_sim.mat
save Janes_all_sim.mat
```

Note that now `Janes_all_sim.mat` contains all above discussed features of FAST. However, if she just starts `FAST` and uses **New Simulation** then the obstacles, reference signal and changes will not be loaded. For example to load the changes she needs, as before, to check the box under **Formation Changes** and select `Janes_all_sim.mat`

References

[PPW09] U. Pilz, A. Popov, and H. Werner. Robust controller design for formation flight of quad-rotor helicopters. In *Proc. 48th IEEE Conference on Decision and Control*, pages 8322–8327, 2009.

[PW09] A. Popov and H. Werner. A robust control approach to formation control. In *Proc. of the European Control Conference*, pages 4428–4433, 2009.

[Ros09] H. Rose. Design and implementation of a simulation environment for multi-agent system formation control. Diplomarbeit, Technische Universität Hamburg-Harburg, 2009.