**The cMANNgo Program**

Ronald J. Mintz

August 7, 2018

OVERALL NOTES AND TO DO:

1. Re order a bit to this (notice that your sections below are included here):

--Intro and Rationale [this is new, MARK will do this part]

--Simulation Structure and Dynamics (I've put in a heading where this might be, so its old)

--Setting Up the Prototypes (old from below)

--On Each Tick (old from below)

--Example simple init script (annotated) (this is new) showing how to start up a simple case that is just one point in the parameter space, so people can get to get used to the system.

--Output files

--Experiments and Parameter Combinations (old from below)

2. Look through the doc for RED stuff and address.

MUST ADD:

--A link to Dan Chen's setup for LENS will be essential [MARK WILL DO THIS LATER]

--MAYBE MAKE ONE LENS INSTANCE LIKE IN MATRIX.

Introduction and Rationale

An introductory paragraph that provides the motivation for cManngo (ORR WILL DO THIS)..

[In the "Intro.." we need a few sentences about the software specs including stuff about LENS C libs ].

E.G...."So, to these ends, we built a simulation platform with the following specifications: .."

Simulation Structure and Dynamics

The system is composed of a network of interacting agents to model social communication. Each agent is internally processed by a LENS [ORR: we should put a reference/link to LENS here] neural network containing an input, output, and hidden layer. [See Lens, the light efficient network simulator by Douglas Rohde.

<http://tedlab.mit.edu/~dr/Lens/> for detail on LENS.]

Each layer is composed of an array of features, each of which is represented by a real number. The neural network of each agent maps an input array of features to an output array of features. There is one hidden layer in between which is mostly transparent to the system. The output array, which describes the last state of each agent, can be used as the input to another agent to which it is connected in the network. The system contains the following items.

1. Agents
2. Features
3. Network specifying the edges connecting the agents. The network is generated and specified using igraph. See reference 2 for details of igraph.
4. Prototypes that provide initial input and possible subsequent input to the agents.
5. Time ticks. At each tick, data is input to an agent from either another agent or one of the prototypes.

A simulation is described by the following parameters

**n\_agents** = number of agents in system

**n\_features** = number of features for input to and output from each agent.

**n\_hidden** = number of features in the hidden layer. This is computed by multiplying the parameter **proportion\_hidden** by **n\_features** and rounding to an integer.

**n\_ticks** = number of ticks in the runs. This is generally proportional to **n\_agents**.

**n\_runs** = number of runs with identical parameters. The data in each run will differ because of pseudo-randomness.

**proto\_p\_on**, **proto\_p\_flip**, and **item\_p\_flip** are used to generate the prototypes, as explained in the next section.

**SOCIAL\_PROB\_ALGORITHM** and **social\_prob\_parameter** are used to randomly determine whether the input to an agent on each specific tick will come from random modification of its prototype or the output of another agent which connects to it in the network.

**graph\_type** specifies the igraph algorithm used to randomly generate the graph which describes the connections among the network of agents. It is currently either IGRAPH\_WATTS\_STROGATZ or IGRAPH\_ERDOS\_RENYI (which has two subtypes). This graph is generated by igraph at the start of each run. Its edges determine the agents that can send their output as input to another agent. The parameters **NEIGHBORHOOD** and **PROB\_REWIRE** are used in the generation of a Watts Strogatz graph and correspond to the arguments **nei** and **p** in the igraph reference manual’s igraph\_watts\_strogatz\_game function. The edges of the graph are stored into the file **connections\_*run#.txt*** where ***run#*** is the number of the run that generated the graph. The graphs for each run will differ due to randomness in their construction even though the parameters are the same in each run.

These parameters will remain constant over the series of runs, but version 7 and later versions provide a way to automate the setting up of differing parameter combinations for each of several series of runs to greatly reduce the work of manually entering the parameters for each series of runs. This technique will be described later.

Setting up the Prototypes

Each prototype is an array of **n\_features** numbers, each of which is randomly determined to be either zero or one. At the start of each run, the program creates a single uber (main) prototype, each of whose features is independently set to 1 with probability **proto\_p\_on**, otherwise 0. We derive a separate prototype for each agent from a random distortion of the uber prototype. Each feature is set to either its value in the uber prototype, or a new randomly derived value which is 1 with probability **proto\_p\_on**, otherwise 0. This choice of these two alternatives is again random. The probability of choosing the new randomly derived value alternative is **proto\_p\_flip**. [WE SHOULD REFERENCE PLAUT's PAPER FOR THIS, I THINK HE USES THE SAME ALGORITHM.]

When we decide to obtain the input for an agent from its prototype [during the simulation runtime or in the pretraining], the agent’s prototype is again randomly distorted for this specific use in a similar manner to the above, but the probability of choosing the new randomly derived value alternative is **item\_p\_flip**. [CAN WE MAKE A GRAPHIC FOR THIS WHOLE PROTOTYPE BUSINESS?].

[We need a higher level of description here and maybe a graphic that conveys the larger picture; e.g., the uber protote is set so that there is a basis of comparison among initial conditions and, sometimes, inputs, the proto\_p\_on derives a variation on the uber for each agent that is fixed while the proto\_p\_flip captures the variation within an agents own prototype, when required. This could be described using distributions in feature space.]

Before the agents start to interact, they must all have output values, computed by Lens from respective input values. The initial input value for each agent is a distortion of its agent specific prototype, computed as described in the last paragraph.

On Each Tick

For each tick, an edge is randomly chosen from the graph. This identifies a sender and receiver agent. The boolean valued function **usingSocialForInput(tick**) is then called to decide whether the input of the receiver will be the output of the sender, or whether it will be taken from a distortion of the receiver’s prototype instead. This function makes a random decision, choosing the output of the sender with probability determined by the **SOCIAL\_PROB\_ALGORITHM** which generally uses one parameter, **social\_prob\_parameter**. We may choose **SOCIAL\_PROB\_ALGORITHM** to be any of the following algorithms.

* **constant**. The probability of using the output of the sender for input is then **social\_prob\_parameter** regardless of the tick number.
* **linear**. The probability of using the output of the sender for input is the ratio of the tick number to **n\_ticks**, the total number of ticks.
* **logistic\_increasing**.
* **logistic\_decreasing**.

The function **usingSocialForInput(tick**) then makes a random decision with the probability computed above.

We then use the function **computeOutputs** to call Lens to compute the outputs for the receiver agent from either the sender agent’s outputs or the distortion of the receiver’s prototype.

Setting the Parameter Values

For versions 6 and below, the values of the parameters described in the section “Simulation Structure and Dynamics” are hardcoded in the program social.c, usually using #define. Version 7 allows multiple values to be assigned to certain parameters in order to enable multiple runs with different values. This is explained in the section “Experiments and Parameter Combinations.” It therefore uses a different format for these parameters. Versions 6 and below include the following code, which is edited for the parameter values.

#define N\_AGENTS 10

#define N\_FEATURES 6

// N\_FEATURES was formerly N\_INPUTS = N\_OUTPUTS

#define PROPORTION\_HIDDEN 0.3

#define N\_HIDDEN ((int)round((N\_FEATURES) \* (PROPORTION\_HIDDEN)))

#define N\_TICKS 1000

#define N\_RUNS 2

// To seed random number generator based on time, specify a negative number for SEED,

// otherwise SEED is used as the seed for all random number generators, including igraph.

#define SEED 12345

#define PROTO\_P\_ON 0.5

#define PROTO\_P\_FLIP 0.2

#define ITEM\_P\_FLIP 0.1

#define SOCIAL\_PROB\_ALGORITHM "constant"

#define SOCIAL\_PROB\_PARAMETER 0.2

#define LEARNING\_RATE 0.05

#define MOMENTUM 0.9

// igraph parameters for IGRAPH\_WATTS\_STROGATZ

#define NEIGHBORHOOD 4

#define PROB\_REWIRE 0.10

#define CMDLEN 100000

#define DISPLAY\_TO\_SCREEN 0

#define SAVE\_WEIGHTS 0

#define STORE\_AGENT\_CONNECTIONS 1

#define OMIT\_ROWS\_FOR\_AGENTS\_NOT\_UPDATED 1

// defining CONNECTION\_TYPE to be an IGRAPH.\_.. causes agent connection to be represented by edges in igraph network

#define USE\_IGRAPH

…

#ifdef USE\_IGRAPH

typedef enum {IGRAPH\_WATTS\_STROGATZ, IGRAPH\_ERDOS\_RENYI} Igraph\_type;

igraph\_t graph;

Igraph\_type graphType = IGRAPH\_WATTS\_STROGATZ;

void initAgentConnections(int runNum)

{

// The actions depending on the type of igraph are contained within this switch statement.

// This switch statement sets up the type of graph selected above.

switch(graphType)

{

case **IGRAPH\_WATTS\_STROGATZ**:

{

int error1 = igraph\_watts\_strogatz\_game(&graph, 1, N\_AGENTS, **NEIGHBORHOOD**, **PROB\_REWIRE**, 0, 0);

int error2 = igraph\_to\_directed(&graph, IGRAPH\_TO\_DIRECTED\_MUTUAL); // creates bi-directional graph

...

break;

}

case **IGRAPH\_ERDOS\_RENYI**:

{

igraph\_erdos\_renyi\_t graphSubtype = **IGRAPH\_ERDOS\_RENYI\_GNM**;

igraph\_real\_t **p\_or\_m** = round(0.05 \* N\_AGENTS \* N\_AGENTS);

int error1 = igraph\_erdos\_renyi\_game(&graph, graphSubtype, N\_AGENTS, p\_or\_m, 0, 0);

int error2 = igraph\_to\_directed(&graph, IGRAPH\_TO\_DIRECTED\_MUTUAL); // creates bi-directional graph

...

break;

}

...

}

...

#endif

In the above code, igraph is used to define the connections between agents. This allows small-world networks to be easily implemented. We assume that igraph is used, but the code allows connections to be defined without the use of igraph if desired, in which case we would omit the line #define USE\_IGRAPH.

When igraph is used, the user has the choice of using the graph types IGRAPH\_WATTS\_STROGATZ or IGRAPH\_ERDOS\_RENYI. IGRAPH\_WATTS\_STROGATZ uses parameters NEIGHBORHOOD and PROB\_REWIRE. IGRAPH\_ERDOS\_RENYI must specify a subtype IGRAPH\_ERDOS\_RENYI\_GNM or IGRAPH\_ERDOS\_RENYI\_GNP as well as the parameter p\_or\_m. See reference 2 for more detail.

After entering the parameter values into social.c, type **make social** to compile the code and generate an executable **social**. Then type **./social** to run the program. **make social** uses the file **Makefile** which may have to be edited to reference the locations of Lens and igraph appropriately,

See the section Experiments and Parameter Combinations for an illustration of how to specify the values of certain parameters that may assume either a single value or multiple values to generate multiple runs.

Output Files

The file **parameters\_#.txt** (where # denotes the run number (out of N\_RUNS identical parameter sets) shows the parameters that were used in the run. parameters\_1, parameters\_2, etc. should all be identical. Each of these files looks like

n\_agents 10

n\_ticks 1000

n\_features 6

proportion\_hidden 0.300000

n\_hidden 2

N\_RUNS 2

SEED 12345

proto\_p\_on 0.500000

proto\_p\_flip 0.200000

item\_p\_flip 0.100000

SOCIAL\_PROB\_ALGORITHM constant

social\_prob\_parameter 0.200000

LEARNING\_RATE 0.050000

MOMENTUM 0.900000

GRAPH\_TYPE IGRAPH\_WATTS\_STROGATZ

NEIGHBORHOOD 4

PROB\_REWIRE 0.100000

The file **connections\_#.txt** shows the connections (sender, receiver) between the agents, which are edges in the graph. Because of randomness in the construction of the graph, **connections\_0.txt** will generally differ from **connections\_1.txt** even though the parameter set is identical. These files have the form

1 0

6 1

3 2

9 4

5 4

6 5

7 6

8 7

9 8

9 0

2 0

8 0

3 0

6 0

4 0

5 0

9 1

3 1

8 1

4 1

7 1

5 1

6 2

9 2

5 2

…

where the first column is the number of the sending agent and the second column is the number of the receiving agent.

The file **prototypes\_#.txt** shows the Uber prototype and the prototype of each agent. Because of randomness in the generation of all prototypes, prototypes\_0.txt will generally differ from prototypes\_1.txt. The first column in the table indicates the agent number or ‘**U**’ for the Uber prototype. If there are 10 agents, each with 6 features, prototypes\_#.txt has the following format.

U 1.00 0.00 1.00 0.00 0.00 0.00

0 1.00 0.00 1.00 0.00 0.00 0.00

1 1.00 0.00 1.00 0.00 0.00 0.00

2 1.00 0.00 1.00 0.00 0.00 0.00

3 1.00 0.00 1.00 1.00 0.00 0.00

4 1.00 0.00 1.00 0.00 1.00 0.00

5 1.00 0.00 1.00 1.00 0.00 0.00

6 1.00 0.00 0.00 0.00 0.00 1.00

7 1.00 0.00 1.00 0.00 0.00 0.00

8 1.00 0.00 1.00 0.00 0.00 0.00

9 1.00 0.00 1.00 0.00 0.00 0.00

The file **history\_#.txt** shows the inputs and outputs of the receiving agent on each tick. It is usually run with the option OMIT\_ROWS\_FOR\_AGENTS\_NOT\_UPDATED = 1 so that the data for agents not modified is not shown in order to keep the file size more reasonable. It tends to be very large in most experiments. The first column is the tick number. The second column is the number of the receiving agent. If this agent is the receiving agent, the third column is flagged with **1** and the fourth column indicates the number of the sending agent or ‘**P**’ if the input is from a distortion of the receiving agent’s prototype. The reason for flagging rows for receiving agents with 1 is that rows for all agents would be shown if run with the option OMIT\_ROWS\_FOR\_AGENTS\_NOT\_UPDATED = 0, so we need to know which is the receiving agent. The next n\_features columns are the values of the inputs to the receiving agent. The following n\_features columns are the values of the outputs from the agent.

Tick 0 represents the pretraining of each agent, where all input is from a distortion of the agent’s prototype in order to supply an initial output value for each agent. There is therefore a row for tick 0 with each agent number. It is followed by n\_features columns with the values of the inputs to the agent. The following n\_features columns are the values of the outputs from the agent.

**history\_#.txt** thus has the format:

<tick#> <agent#> <1 if receiving agent> <sending agent#> <6 inputs> <6 outputs>

**0** 0 - - 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.761705 0.572299 0.840292 0.572993 0.601818 0.790051

**0** 1 - - 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.601359 0.482714 0.481729 0.593022 0.348901 0.272290

**0** 2 - - 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.250172 0.189055 0.487898 0.537012 0.624166 0.548238

**0** 3 - - 1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.660231 0.504188 0.352391 0.570726 0.224996 0.489200

**0** 4 - - 1.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.507182 0.559972 0.433009 0.357234 0.161730 0.521287

**0** 5 - - 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.268650 0.779824 0.245190 0.549229 0.547758 0.660534

**0** 6 - - 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 0.388153 0.445261 0.832266 0.633165 0.330773 0.675508

**0** 7 - - 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.287800 0.525446 0.260016 0.437109 0.168835 0.503582

**0** 8 - - 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.612038 0.378353 0.806737 0.382069 0.507050 0.141030

**0** 9 - - 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.628232 0.310701 0.126352 0.534278 0.380735 0.764262

**1** 3 1 9 0.628232 0.310701 0.126352 0.534278 0.380735 0.764262 0.602552 0.515885 0.372218 0.541782 0.225686 0.509449

**2** 6 1 1 0.601359 0.482714 0.481729 0.593022 0.348901 0.272290 0.423917 0.434510 0.782201 0.589587 0.385372 0.679579

**3** 0 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.760435 0.576394 0.828654 0.567882 0.559542 0.779967

**4** 2 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.257751 0.188198 0.494661 0.530085 0.617603 0.540781

**5** 1 1 P 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.615012 0.480155 0.484190 0.597543 0.362555 0.255924

**6** 5 1 3 0.602552 0.515885 0.372218 0.541782 0.225686 0.509449 0.298329 0.752709 0.264635 0.558521 0.511963 0.620244

**7** 5 1 P 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.289691 0.761886 0.260343 0.558353 0.525780 0.641780

**8** 0 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.763412 0.561197 0.829857 0.566063 0.543304 0.764629

**9** 3 1 P 1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.619021 0.494986 0.365441 0.526257 0.211892 0.505477

**10** 8 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.620290 0.371192 0.809493 0.373861 0.496827 0.139762

**11** 0 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.767695 0.539593 0.831656 0.557155 0.520525 0.741805

**12** 4 1 P 1.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.513397 0.552533 0.440578 0.353216 0.167953 0.514367

**13** 2 1 9 0.628232 0.310701 0.126352 0.534278 0.380735 0.764262 0.281686 0.179651 0.498985 0.498660 0.595800 0.532315

**14** 2 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.291903 0.186482 0.510692 0.505628 0.589304 0.519242

**15** 6 1 P 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 0.406926 0.437453 0.816367 0.618228 0.325314 0.673349

**16** 3 1 P 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.647887 0.490883 0.345032 0.601401 0.243088 0.483175

**17** 8 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.635728 0.357754 0.814653 0.358514 0.477439 0.137360

**18** 7 1 P 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.295384 0.520751 0.266910 0.431804 0.168549 0.497812

**19** 3 1 P 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.662685 0.471214 0.349754 0.616985 0.245847 0.469008

**20** 0 1 P 1.000000 1.000000 1.000000 0.000000 0.000000 1.000000 0.743935 0.556732 0.771796 0.516537 0.377691 0.701199

…

Experiments and Parameter Combinations

We sometimes want to specify a set of values for each of several parameters and to do a separate run for each possible combination of these parameter values. For example, we might want to run combinations of:

**n\_agents** = 100, 1000, 2000

**n\_features** = 20, 40

**proportion\_hidden** 0.3, 0.5

Each of the other parameters will take on a single value for all runs. Thus, we want to run each of the following parameter combinations:

**n\_agents** **n\_features** **proportion\_hidden**

100 20 0.3

100 20 0.5

100 40 0.3

100 40 0.5

1000 20 0.3

1000 20 0.5

1000 40 0.3

1000 40 0.5

2000 20 0.3

2000 20 0.5

2000 40 0.3

2000 40 0.5

Manually changing the value of each of these parameters for each combination can be a time-consuming and tedious task. I therefore automate this process in versions 7 and above of cMaango. This enables us to specify the set of values to be taken on by each of the three parameters as shown on top of this page. The program will then run each combination of these parameters as a separate run. If **n\_runs** > 1, each of the above 12 parameter combinations will be run **n\_runs** times with the same parameter set.

The parameters for which a vector of values can be supplied are listed in the function **processAllParamCombos()**. The elements of each vector are entered in this function. Define a one element vector if the parameter is to take on only one value. Parameters for which this automated feature is not enabled are specified as #define constants early in the program.

For example, to have **n\_agents** take on the values 100, 1000, and 2000, we set

int v\_n\_agents[] = { 100, 1000, 2000 };

l\_n\_agents is then automatically computed to be 2, the length of this vector.

The code is currently set up to process values for the parameters n\_agents, n\_features, proportion\_hidden, proto\_p\_on, proto\_p\_flip, item\_p\_flip, and social\_prob\_parameter. n\_ticks is then automatically computed from n\_agents. n\_hidden is automatically computed from proportion\_hidden and n\_features. These are currently the only parameters which the version 7 code is set up to process multiple values. Any parameters defined by #define, such as N\_RUNS, currently assume only the value specified in the #define. For allocating the size of arrays, we require n\_agents <= MAX\_AGENTS and n\_features <= MAX\_FEATURES, where the values of MAX\_AGENTS and MAX\_FEATURES are fixed values specified by #define statements.

The code where the user needs to specify the parameter values has the following structure:

int v\_n\_agents[] = { 100, 1000, 2000 }; // vector of values for n\_agents

int l\_n\_agents = sizeof(v\_n\_agents)/sizeof(v\_n\_agents[0]); // length of v\_n\_agents

int i\_n\_agents; // index into v\_n\_agents to identify value being used

int v\_n\_features[] = { 20, 40 };

int l\_n\_features = sizeof(v\_n\_features)/sizeof(v\_n\_features[0]);

int i\_n\_features; // index into v\_n\_features

real v\_proportion\_hidden[] = { 0.3, 0.5 };

int l\_proportion\_hidden = sizeof(v\_proportion\_hidden)/sizeof(v\_proportion\_hidden[0]);

int i\_proportion\_hidden;

real v\_proto\_p\_on[] = { 0.5 };

int l\_proto\_p\_on = sizeof(v\_proto\_p\_on)/sizeof(v\_proto\_p\_on[0]);

int i\_proto\_p\_on;

real v\_proto\_p\_flip[] = { 0.2 };

int l\_proto\_p\_flip = sizeof(v\_proto\_p\_flip)/sizeof(v\_proto\_p\_flip[0]);

int i\_proto\_p\_flip;

real v\_item\_p\_flip[] = { 0.1 };

int l\_item\_p\_flip = sizeof(v\_item\_p\_flip)/sizeof(v\_item\_p\_flip[0]);

int i\_item\_p\_flip;

real v\_social\_prob\_parameter[] = { 0.2 };

int l\_social\_prob\_parameter = sizeof(v\_social\_prob\_parameter)/sizeof(v\_social\_prob\_parameter[0]);

int i\_social\_prob\_parameter;

for (i\_n\_agents = 0; i\_n\_agents < l\_n\_agents; i\_n\_agents++)

{

n\_agents = v\_n\_agents[i\_n\_agents];

n\_ticks = 100 \* n\_agents;

for (i\_n\_features = 0; i\_n\_features < l\_n\_features; i\_n\_features++)

{

n\_features = v\_n\_features[i\_n\_features];

for (i\_proportion\_hidden = 0; i\_proportion\_hidden < l\_proportion\_hidden; i\_proportion\_hidden++)

{

proportion\_hidden = v\_proportion\_hidden[i\_proportion\_hidden];

n\_hidden = (int)round(proportion\_hidden \* (real)n\_features);

...

References

1. Lens, the light efficient network simulator by Douglas Rohde.

<http://tedlab.mit.edu/~dr/Lens/>

1. igraph Reference Manual, by Gábor Csárdi and Tamás Nepusz

<http://igraph.org/c/doc/igraph-docs.pdf>