A Turing Complete SDS

Andrew Owen Martin

23 February 2018

Contents

1 Introduction **2**

2 Preliminary assumptions **2**

3 Just copy NORMA2 **3**

3.1 Two agent solution 3

3.2 Problems with the Two Agent solution 10

4 Multiple agent solution **10**

4.1 Implementation 10

4.2 Multi swarm definitions 12

4.3 Multi-swarm execution 16

5 Multi swarm solution with functional programming **18**

6 Quorum sensing **23**

6.1 Agent activity 24

6.2 Quorum test function 25

6.3 Quorum diffusion function 27

6.4 Quorum microtest 28

6.5 Quorum new hypothesis 30

6.6 Quorum build new solution 31

State	Action	if $x = 0$ goto	if $x > 0$ goto
1	Init	4	2
2	DEC x	3	3
3	INC y	4	2
4	Halt	4	4

Table 1: Norma addition truth table

6.7 Quorum iterate 33

6.8 Quorum execute 36

6.9 OLD Quorum summary pseudocode 39

6.10 Quorum summary pseudocode 40

A Index **41**

B Code Chunks **41**

1 Introduction

Run the code for the Multi-swarm solution with `make code && python multi_swarm_solution.py` and for the Two Agent solution with `make code && python run_two_agent_solution.py`.

2 Preliminary assumptions

To go from something like NORMA2 to a new swarmy implementation, we need to know what things we can and cannot change.

Firstly, when designing a NORMA2 program, to perform addition for example, then an intelligent user still needs to determine the procedure themselves. This means we can get away with manually specifying the shape of the program, and should not expect the intended function to emerge without such careful initialisation. Because of this, I will often refer to the state table which defines a NORMA2 machine for addition (Table 1); in a properly swarmy system, there will not be an explicit state table, but the emergent behaviour will have to be similar to one, and this will largely have to be the result of manual configuration by a user.

Secondly, NORMA2 was given some registers of unlimited size which is fine in practise, but if I were to implement NORMA2, in Python for example, then I would need to include special routines for times when the value of the registers was larger than could fit into memory. This could be a function which stops the NORMA2 procedure before the register values become too big, and informs the user to go and install more memory into the machine. The point is, that any implementation I can do in a few minutes would not be able to perform certain computations over a certain size. I therefore suggest that we should not expect the swarmy implementation to work for all cases either, though in theory it should.

Lastly, the capabilities of NORMA are incrementing, decrementing, branching, and comparison with zero. We will need either the same set of capabilities, or a similarly expressive set.

With these assumptions in mind, lets consider a few approaches.

3 Just copy NORMA2

If I implemented something exactly like NORMA2, then we know for sure that it would work, but we would be criticised for demonstrating something that is neither original or swarmy. So from this basis, what's the minimum change we could do to make a system that would still work, by virtue of its being almost identical to NORMA2, but with some swarmy aspect?

Imagine a swarm of one agent, where their behaviour was entirely determined by the state table. The agent would maintain a state and two register values, and update them according to the states. Again, this would definitely work, but not be sufficciently swarmy, as there is no agent-to-agent interaction, and the behaviour of the agent is explicit and complicated.

So we need more than one agent, is there any way that two agents could be used? Maybe if one agent simply maintained a boolean state depending on whether it held that $x = 0$. Then you could have one agent maintaining the registers and state, and branching depending on the state of the other agent. This could also certainly be made to work, but now we have the criticism that the two distinct types of agent makes this more of a multi-agent system rather than a swarm intelligence system.

So could it be done with two identical agents? One would need to maintain a state and the other would have to find the next state. The agent which found the next state would have to deactivate the other agent so the newly inactive agent may find the next state. They would go on leap-frogging each other. Let's see if that's at all viable.

3.1 Two agent solution

First we'll import some bits, and define an Agent as something which has an activity and a hypothesis, and a 'Step' which defines a state of a computational procedure, representing the value of the X and Y registers and the current state it is in.

3 ⟨*two-agent-solution.py* 3⟩≡

```
import random
from collections import namedtuple

Step = namedtuple('Step',['x','y','state'])
```

```
class Agent:
    def __init__(self):
        self.active = False
        self.hyp = None
```

This definition is continued in chunks 4–8.
Root chunk (not used in this document).

Then we'll define a swarm as two of those agents, and define the function for generating new hypothesis, which in this case returns the initial Step, where $X = 2$, $Y = 1$, in State 1.

4 ⟨*two-agent-solution.py* 3⟩+≡

```
swarm = tuple(Agent() for _ in range(2))

def initial_hyp():
    return Step(x=3,y=2,state=1)
```

Now we need a diffusion phase. I've chosen to disallow self-selection, and use standard passive diffusion, and hypothesis transmission error, where the registers have a small chance of being perturbed and the state is chosen entirely random.

I previously had used context-sensitive to ensure both agents wouldn't remain active, but that event never occurs.

5 ⟨*two-agent-solution.py* 3⟩+≡
```
def diffuse(swarm):
    for agent_num, agent in enumerate(swarm):

        # No self selection.
        # The polled agent is always the other agent.
        other_agent = swarm[(agent_num+1)%2]

        if not agent.active:

            if other_agent.active: # copy hyp noisily

                x, y, state = other_agent.hyp

                if random.random() < 0.3:

                    x += random.choice([-1,1])

                if random.random() < 0.3:

                    y += random.choice([-1,1])

                agent.hyp = Step(x, y, state=random.randint(1,4))

            else:

                agent.hyp = initial_hyp()
```

That all seems fairly safe, I should point out that the randomisation of state in the transmission error will be mitigated if each agent existed within a swarm that defined a single state. To clarify this, rather than a single homogeneous swarm, imagine a swarm of agents in each row of the state table, then each swarm would "know" what they were hypothesising as the next state.

But now I need to determine a test phase. It would require checking if the other agent is active and if so, that your hypothesis is a valid transition (see function "is_valid$_7$"). There also needs to be the extra feature that the agents remain active once they've become active, and the surpress the activity of the other agent.

6 ⟨*two-agent-solution.py* 3⟩+≡
```
def test():

    for agent_num, agent in enumerate(swarm):

        # Active agents do nothing, and stay active
        if not agent.active:

            other_agent = swarm[(agent_num+1)%2]

            if other_agent.active:

                # Check if transition is valid.

                valid = is_valid7(agent.hyp, other_agent.hyp)

                agent.active = valid

                if valid:

                    other_agent.active = False

            else:

                # Both agents are inactive.
                # Go active if you have the initial state.
                # This only runs once.

                agent.active = agent.hyp == initial_hyp()
```
Uses is_valid 7.

Here's the `is_valid`$_7$ function, which implements the state table manually.

7 ⟨*two-agent-solution.py* 3⟩+≡

```
def is_valid7(hyp, prev_hyp):

    other_x, other_y, other_state = prev_hyp

    x, y, state = hyp

    if other_state == 1:

        if other_x == 0:

            return x == other_x and y == other_y and state == 4

        else:

            return x == other_x and y == other_y and state == 2

    elif other_state == 2:

        return x == other_x-1 and y == other_y and state == 3

    elif other_state == 3:

        if other_x == 0:

            return x == other_x and y == other_y+1 and state == 4

        else:

            return x == other_x and y == other_y+1 and state == 2

    else: # other_state == 4:

        return False
```

Defines:

`is_valid`, used in chunk 6.

Now to add a loop that does intialise, test, diffuse and halt. We'll halt when any active agent is in halt state 4.

8 ⟨*two-agent-solution.py* 3⟩+≡

```
def run():
    # Initialise
    for agent in swarm:
        agent.hyp = initial_hyp()

    previously_active_agent = 1

    for iteration in range(1000):

        test()

        diffuse(swarm)

        # Print to console when the inactive agent becomes active.
        active_agent = [agent.active for agent in swarm].index(True)

        if False or not (active_agent == previously_active_agent):

            print(
                format(iteration,'4'),
                [(agent.hyp,agent.active) for agent in swarm],)

            previously_active_agent = active_agent

        # Halting.
        if any(agent.hyp.state == 4 for agent in swarm if agent.active):

            print(
                'Answer is',
                next(
                    agent.hyp
                    for agent
                    in swarm
                    if agent.active and agent.hyp.state == 4).y,
                'in',iteration,'iterations.')

            break
    else:

        print('no convergence')
```

Here's a script to run the two agent solution. It works, but there are some potentially problematic aspects to it.

9 ⟨*run-two-agent-solution.py* 9⟩≡
```
import two_agent_solution

two_agent_solution.run()
```
Root chunk (not used in this document).

3.2 Problems with the Two Agent solution

1. The complex logic in the test for validation.
2. The fact that there are only two agents, there's nothing swarmy.
3. Only inactive agents perform the test phase.

4 Multiple agent solution

The idea is to have one swarm for every state in the NORMA2 state table, plus two for every unique conditional branch destination.

Each swarm will also have a boolean convergence flag, which is initially false.

Initialisation could be achieved either by setting all the agents in the initial swarm to certain register values, and setting the convergence flag, or setting a single agent active with certain register values and waiting for convergence. Agents in the initial swarm will be set active and given register values.

Any swarm which has not converged will be polling connected swarms and diffusing perturbed hypotheses, which it will test as valid by checking that a random agent from the previous swarm is converged and the registers make sense with the polled agents registers.

When a swarm has full activity it sets its convergence flag, inviting activity from the subsequent swarms, and surpresses all activity in previous swarms by switching off its convergence flag. Convergence flags therefore surpress processing.

4.1 Implementation

Let's start off with some imports.

10 ⟨*multi-swarm-solution.py* 10⟩≡
```
import random
import sds
from collections import namedtuple
import functools
import sys
import itertools
import json

XY = namedtuple('XY',['x','y'])

def agent_repr(self):
    if self.active:
        return "{h}".format(h=self.hypothesis)
    else:
        return "inactive"
sds.Agent.__repr__ = agent_repr
```

This definition is continued in chunks 11–16, 18–22, 24b, 25, 27, 29–31, and 33–36.
Root chunk (not used in this document).

Now I'll define some swarms, right now I'll just initialise them, I'll link them up later, as its much easier to link an acyclic graph once all nodes have been created.

I'm also including a 'converged' flag for each swarm, and rather than just having a raw boolean `'False'`, I've made it a boolean in a list `'[False]'` so I can zip the list and have something I can modify in place.

At the end of this chunk 'swarms' is a list of (swarm name, agent list, converged flag) tuples.

11a ⟨*multi-swarm-solution.py* 10⟩+≡

```
agent_count = 5
swarm_names = ('init','xgt0','decx','incy','xeq0','halt')
swarms = {
    swarm_name: {
        'converged': False,
        'agents': [sds.Agent() for _ in range(agent_count)]
    }
    for swarm_name
    in swarm_names
}
```

Now initialise all the agent in the first swarm as active at the initial step.

This action has to be considered carefully as it must be the case that this swarm never converges again. Once the convergence flag is on, it will stop processing, once the flag goes off again, there must never be any activity, so activity can't just come from the microtests. I therefore suggest that each agent in the init swarm selects an agent in the init swarm.

11b ⟨*multi-swarm-solution.py* 10⟩+≡

```
for agent in swarms['init']['agents']:
    agent.hypothesis = XY(x=10,y=10)
    agent.active = True

swarms['init']['converged'] = True
```

4.2 Multi swarm definitions

I'll define everything manually, but afterwards I should be able to make a sexy functional programming solution.

This is for the init swarm.

12a ⟨*multi-swarm-solution.py* 10⟩+≡
```
def init_hyp(rng=random):
    #print('init hyp runs')
    return XY(x=1000,y=1000)

def init_test(hyp):
    #print('init test runs')
    polled_agent = random.choice(swarms['init']['agents'])
    #return polled_agent.hypothesis == XY(x=3,y=2)
    return polled_agent.active
```

Before doing any of the other functions, we have to have a function to probabilistically perturb a register.

12b ⟨*multi-swarm-solution.py* 10⟩+≡
```
def perturb(register): #PERTURB
    return register + random.choice((1,0,0,0,-1))
```

This is for '$x > 0$' swarm.

12c ⟨*multi-swarm-solution.py* 10⟩+≡
```
def xgt0_hyp(rng=random):
    for swarm_name in ('init','incy'):
        if swarms[swarm_name]['converged']:
            polled = rng.choice(swarms[swarm_name]['agents'])
            polled_x, polled_y = polled.hypothesis
            return XY(x=perturb(polled_x),y=perturb(polled_y))
    else:
        return XY(x=0,y=0)

def xgt0_test(hyp):
    for swarm_name in ('init','incy'):
        if swarms[swarm_name]['converged']:
            polled_agent = random.choice(swarms[swarm_name]['agents'])
            break
    else:
        return False
    if polled_agent.active:
        polled_x, polled_y = polled_agent.hypothesis
        return hyp.x == polled_x and hyp.y == polled_y and hyp.x > 0
    else:
        raise RuntimeError('inactive agent polled')
```

This is for ‘Decrement X’ swarm.

13a ⟨*multi-swarm-solution.py* 10⟩+≡
```
def decx_hyp(rng=random):
    for swarm_name in ('xgt0',):
        if swarms[swarm_name]['converged']:
            polled = rng.choice(swarms[swarm_name]['agents'])
            polled_x, polled_y = polled.hypothesis
            return XY(x=perturb(polled_x),y=perturb(polled_y))
    else:
        return XY(x=0,y=0)

def decx_test(hyp):
    if swarms['xgt0']['converged']:
        polled_agent = random.choice(swarms['xgt0']['agents'])
        if polled_agent.active:
            polled_x, polled_y = polled_agent.hypothesis
            result = hyp.x == polled_x-1 and hyp.y == polled_y
            return result
        else:
            raise RuntimeError('inactive agent polled')
    else:
        print('dec x failed test as xgt0 is not converged')
        return False
```

This is for ‘Increment Y’ swarm...

13b ⟨*multi-swarm-solution.py* 10⟩+≡
```
def incy_hyp(rng=random):
    for swarm_name in ('decx',):
        if swarms[swarm_name]['converged']:
            polled = rng.choice(swarms[swarm_name]['agents'])
            polled_x, polled_y = polled.hypothesis
            return XY(x=perturb(polled_x),y=perturb(polled_y))
    else:
        return XY(x=0,y=0)

def incy_test(hyp):
    if swarms['decx']['converged']:
        polled_agent = random.choice(swarms['decx']['agents'])
        if polled_agent.active:
            polled_x, polled_y = polled_agent.hypothesis
            result = hyp.x == polled_x and hyp.y == polled_y+1
            return result
        else:
            raise RuntimeError('inactive agent polled')
    else:
        print('incy failed test as decx is not converged')
        return False
```

This is for ‘$x = 0$’ swarm.

14a ⟨*multi-swarm-solution.py* 10⟩+≡
```
def xeq0_hyp(rng=random):
    for swarm_name in ('init','incy'):
        if swarms[swarm_name]['converged']:
            polled = rng.choice(swarms[swarm_name]['agents'])
            polled_x, polled_y = polled.hypothesis
            return XY(x=perturb(polled_x),y=perturb(polled_y))
    else:
        return XY(x=0,y=0)

def xeq0_test(hyp):
    for swarm_name in ('init','incy'):
        if swarms[swarm_name]['converged']:
            polled_agent = random.choice(swarms[swarm_name]['agents'])
            break
    else:
        return False
    if polled_agent.active:
        polled_x, polled_y = polled_agent.hypothesis
        return hyp.x == polled_x and hyp.y == polled_y and hyp.x == 0
    else:
        raise RuntimeError('inactive agent polled')
```

This is for the halt swarm.

14b ⟨*multi-swarm-solution.py* 10⟩+≡
```
def halt_hyp(rng=random):
    for swarm_name in ('xeq0',):
        if swarms[swarm_name]['converged']:
            polled = rng.choice(swarms[swarm_name]['agents'])
            polled_x, polled_y = polled.hypothesis
            return XY(x=perturb(polled_x),y=perturb(polled_y))
    else:
        return XY(x=0,y=0)

def halt_test(hyp):
    for swarm_name in ('xeq0',):
        if swarms[swarm_name]['converged']:
            polled_agent = random.choice(swarms[swarm_name]['agents'])
            break
    else:
        return False
    if polled_agent.active:
        polled_x, polled_y = polled_agent.hypothesis
        return hyp.x == polled_x and hyp.y == polled_y
    else:
        raise RuntimeError('inactive agent polled')
```

Here's a data structure to keep it all together.

15 ⟨*multi-swarm-solution.py* 10⟩+≡

```
solution = [
    ['init',swarms['init'],init_test,init_hyp,[]],
    ['xgt0',swarms['xgt0'],xgt0_test,xgt0_hyp,['init','incy']],
    ['decx',swarms['decx'],decx_test,decx_hyp,['xgt0']],
    ['incy',swarms['incy'],incy_test,incy_hyp,['decx']],
    ['xeq0',swarms['xeq0'],xeq0_test,xeq0_hyp,['init','incy']],
    ['halt',swarms['halt'],halt_test,halt_hyp,['xeq0']],
]
```

4.3 Multi-swarm execution

16 ⟨*multi-swarm-solution.py* 10⟩+≡
```
def fully_active(swarm):
    result = all(x.active for x in swarm)
    return result

def iterate(solution):
    for swarm_name, swarm, microtest, hyp_function, previous_swarms in solution:
        if not swarm['converged']:
            #print('processing',swarm_name)
            # if swarm is not converged
            sds.run(
                swarm=swarm['agents'],
                microtests=[microtest],
                random_hypothesis_function=hyp_function,
                max_iterations=100,
                halting_function=fully_active,
                halting_iterations=10,
                report_iterations=None,
            )
            if not all(agent.active for agent in swarm['agents']):
                #raise RuntimeError('I thought it would all be active by now')
                #print(swarm_name,"didn't converge")
                pass
            else:
                swarm['converged'] = True
                print(swarm_name,'is now converged at',swarm['agents'][0].hypothesis)
                if swarm_name == 'halt':
                    print('convergence in the halt swarm halts the process')
                    return True
                for swarm_name, swarm, _, _, _ in solution:
                    for prev_swarm_name in previous_swarms:
                        if prev_swarm_name == swarm_name:
                            if swarm['converged']:
                                swarm['converged'] = False
                                print(prev_swarm_name,'is no longer converged')
                                for agent in swarm['agents']:
                                    agent.active = False
                                    agent.hypothesis = None

    #print('not halted yet')
    return False

def do_naive_multiswarm_solution():

    for num, swarm_name in enumerate(swarm_names,start=1):
        print(num, swarm_name, swarms[swarm_name])

    while True:
        if iterate(solution):
            print('halted properly')
            break

    print('done. Answer is',swarms['halt']['agents'][0].hypothesis.y)

#do_naive_multiswarm_solution()
```

5 Multi swarm solution with functional programming

I'll carry on editing the same file as there will be a lot of code sharing.

I need to make some 'intended action' functions to save me typing five different sets of logic.

18 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def decrementX18(hyp):
    hyp_x, hyp_y = hyp
    return XY(hyp_x-1, hyp_y)

def decrementY18(hyp):
    hyp_x, hyp_y = hyp
    return XY(hyp_x, hyp_y-1)

def incrementX18(hyp):
    hyp_x, hyp_y = hyp
    return XY(hyp_x+1, hyp_y)

def incrementY18(hyp):
    hyp_x, hyp_y = hyp
    return XY(hyp_x, hyp_y+1)

def noop18(hyp):
    hyp_x, hyp_y = hyp
    return XY(hyp_x, hyp_y)

def Xeq018(x):
    return x == 0

def Xgt018(x):
    return x > 0

def Xdontcare18(x):
    return True
```

Defines:
- `decrementX`, used in chunk 20b.
- `decrementY`, used in chunk 20b.
- `incrementX`, used in chunk 20b.
- `incrementY`, used in chunk 20b.
- `noop`, used in chunks 19 and 20b.
- `Xdontcare`, used in chunk 20b.
- `Xeq0`, used in chunk 20b.
- `Xgt0`, used in chunks 19 and 20b.

This is a function to make all the test function for all the swarms.

19 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def make_test_fun19(swarms, linked_swarms,action,x_condition,rng=random):
    def generic_test_fun(
        swarms,
        previous_swarm_names,
        intended_action_function,
        x_condition_function,
        rng,
        hyp,
    ):
        for swarm_name in previous_swarm_names:
            if swarms[swarm_name]['converged']:
                polled_agent = rng.choice(swarms[swarm_name]['agents'])
                break
        else:
            return False
        if polled_agent.active:
            polled_x, polled_y = intended_action_function(polled_agent.hypothesis)
            return (
                hyp.x == polled_x
                and hyp.y == polled_y
                and x_condition_function(hyp.x))
        else:
            raise RuntimeError('inactive agent polled')

    return functools.partial(
        generic_test_fun,
        swarms,
        linked_swarms,
        action,
        x_condition,
        rng,)

#foo = make_hyp_fun20a(linked_swarms=['init','incy'],action=noop18,x_condition=Xgt018)
```

Defines:
`make_test_fun`, used in chunk 21.
Uses `make_hyp_fun` 20a, `noop` 18, and `Xgt0` 18.

This is a function to make all the hyp functions for all the swarms.

20a ⟨*multi-swarm-solution.py* 10⟩+≡
```
def make_hyp_fun20a(swarms, linked_swarms):
    def generic_hyp_fun(swarms, linked_swarms,rng=random):
        for swarm_name in linked_swarms:
            if swarms[swarm_name]['converged']:
                polled = rng.choice(swarms[swarm_name]['agents'])
                polled_x, polled_y = polled.hypothesis
                return XY(x=perturb(polled_x),y=perturb(polled_y))
        else:
            return XY(x=0,y=0)
    return functools.partial(generic_hyp_fun, swarms, linked_swarms)
```
Defines:
make_hyp_fun, used in chunks 19 and 21.

Here's are the functions to make the building blocks.

20b ⟨*multi-swarm-solution.py* 10⟩+≡
```
swarm_types = {
    'init':(noop18,Xdontcare18),
    'xgt0':(noop18,Xgt018),
    'xeq0':(noop18,Xeq018),
    'incy':(incrementY18,Xdontcare18),
    'decy':(decrementY18,Xdontcare18),
    'incx':(incrementX18,Xdontcare18),
    'decx':(decrementX18,Xdontcare18),
    'halt':(noop18,Xdontcare18),
}
```
Uses decrementX 18, decrementY 18, incrementX 18, incrementY 18, noop 18, Xdontcare 18, Xeq0 18, and Xgt0 18.

Which can be used with a state table.

20c ⟨*multi-swarm-solution.py* 10⟩+≡
```
state_table = [
    ('A-init','init',['halt']),
    ('B-xgt0','xgt0',['A-init','D-incy']),
    ('C-decx','decx',['B-xgt0']),
    ('D-incy','incy',['C-decx']),
    ('E-xeq0','xeq0',['A-init','D-incy']),
    ('halt','halt',['E-xeq0']),
]
```

Now let's try to build a multi swarm solution from the table.

21 ⟨*multi-swarm-solution.py* 10⟩+≡
```
def build_solution21(state_table,initial_x,initial_y):
    swarms = {
        name: {
            'converged': False,
            'agents': [sds.Agent() for _ in range(agent_count)]
        }
        for name, swarm_type, linked_swarms in state_table
    }

    solution = []

    for name, swarm_type, linked_swarms in state_table:

        if swarm_type == 'init':
            swarm = swarms[name]
            for agent in swarm['agents']:
                agent.hypothesis = XY(x=initial_x,y=initial_y)
                agent.active = True
            swarm['converged'] = True

        action, x_condition = swarm_types[swarm_type]

        test_fun = make_test_fun19(swarms, linked_swarms,action,x_condition,random)
        hyp_fun = make_hyp_fun20a(swarms, linked_swarms)

        solution.append((name,swarms[name],test_fun, hyp_fun, linked_swarms))

    return solution
```
Defines:
  `build_solution`, used in chunk 22.
Uses `make_hyp_fun` 20a and `make_test_fun` 19.

22 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def do_functional_multiswarm_solution():

    functional_solution = build_solution21(state_table,initial_x=1000,initial_y=1000)

    while True:
        if iterate(functional_solution):
            break

    _,halt_swarm,_,_,_ = next(x for x in functional_solution if x[0] == 'halt')
    answer = halt_swarm['agents'][0].hypothesis.y
    print('Answer is',answer)
    #print('done. Answer is',['halt']['agents'][0].hypothesis.y)

#do_functional_multiswarm_solution()
```

Uses `build_solution` 21.

6 Quorum sensing

The previous multi swarm version relied on a flag for each swarm which determined whether or not it was executing. We now hope to replace that flag, with a two-phase behaviour of agents, which implement the effect of the flag once they determine their swarm has converged.

So agents are either (i) polling the previous swarm to check whether their registers are correct, or (ii) remain active, while polling the previous swarm to suppress activity there.

So each agent needs to maintain an internal variable, which is used for quorum sensing.

Here's my take on Bish's pseudocode.

23a ⟨*quorum test phase* 23a⟩≡
```
if agent.quorum_count > quorum_threshold:

    inhibit_mode(linked_swarms)

else:

    polled = random.choice(swarm)

    if (
        polled.quorum_count > quorum_threshold
        and check_registers(agent.hyp, polled.hyp)
    ):

        agent.quorum_count += 1

    else:

        agent.quorum_count = 0
```
Root chunk (not used in this document).

23b ⟨*quorum inhibit mode* 23b⟩≡
```
def inhibit_mode(linked_swarms):
    random_swarm = random.choice(linked_swarms)

    polled = random.choice(swarm)

    polled.quorum_count = 0
```
Root chunk (not used in this document).

24a ⟨*quorum diffusion phase* 24a⟩≡
```
if agent.quorum_count == 0:

    polled = random.choice(swarm)

    if polled.quorum_count > 0:

        agent.hyp = polled.hyp

    else:

        new_hyp()
```
Root chunk (not used in this document).

6.1 Agent activity

We need a new record of activity.

An agent is active, inactive and with a counter for quorum threshold.

24b ⟨*multi-swarm-solution.py* 10⟩+≡
```
Activity = namedtuple("Activity",['active','count'])
```

6.2 Quorum test function

25 ⟨*multi-swarm-solution.py* 10⟩+≡
```
def quorum_test_phase25(
    swarm,
    linked_swarms,
    quorum_threshold,
    swarm_test_function,
    error_prob,
    rng=random
):

    for agent_num, agent in enumerate(swarm):

        random_swarm = random.choice(linked_swarms)

        polled = random.choice(random_swarm)

        if agent.active.count < quorum_threshold:

            if polled.active.count >= quorum_threshold:

                if swarm_test_function(agent, polled):

                    agent.active = Activity(
                        True,
                        agent.active.count,)
                    #agent.active = Activity(
                    #   True,
                    #   agent.active.count+1,)

                else:

                    agent.active = Activity(False, 0)
                    #next_count = agent.active.count - 1
                    #agent.active = Activity(next_count > 0, next_count)

            #else:

            #   next_count = agent.active.count - 1
            #   agent.active = Activity(next_count > 0, next_count)

        else:

            polled.active = Activity(False,0)

        if rng.random() < error_prob:

            new_active = not agent.active.active
            if new_active:

                agent.active = Activity(True, agent.active.count)

            else:

                agent.active = Activity(False, agent.active.count)



    #for agent_num, agent in enumerate(swarm):

    #   if agent.active.count == quorum_threshold:

    #       continue

    #   random_swarm = random.choice(linked_swarms)

    #   polled = random.choice(random_swarm)

    #   # polled is quorate
    #   if polled.active.count == quorum_threshold:

    #       if swarm_test_function(agent, polled):

    #           agent.active = Activity(
    #               True,
    #               agent.active.count,)

    #       else:

    #           # Inactivity due to test failing with a
    #           # polled quorate agent

    #           agent.active = Activity(False, 0)

    #   else:

    #       # Inactivity due to failure to poll a quorate agent.

    #       agent.active = Activity(False, 0)

    #   if rng.random() < error_prob:

    #       agent.active = Activity(True, agent.active.count)
```

Defines:
  quorum_test_phase, used in chunk 33.

6.3 Quorum diffusion function

27 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def quorum_diffusion_phase(
        swarm,
        allow_self_selection,
        quorum_threshold,
        hyp_function,
        error_prob,
        rng,
    ):
        # Quorum diffusion phase
        for agent_num, agent in enumerate(swarm):

            local_polled = random.choice(swarm)

            while (not allow_self_selection) and local_polled is agent:

                local_polled = random.choice(swarm)

            if not agent.active.active:
                # Agent is inactive

                if local_polled.active.active:

                    # Local polled is active, copy hypothesis.
                    agent.hypothesis = local_polled.hypothesis

                    # If local polled is also quorate then go active as well.
                    if local_polled.active.count >= quorum_threshold:

                        agent.active  = Activity(True,agent.active.count)

                else:

                    # Local polled is inactive, random hyp from linked swarm.
                    agent.hypothesis = hyp_function(random)

            elif agent.active.active and agent.active.count < quorum_threshold:
                #Agent is active, but not quorate

                if local_polled.active.active:
                    #Polled is active or quorate

                    if (local_polled.hypothesis == agent.hypothesis):
                        # If registers match then be active and increment count

                        agent.active = Activity(True,min(agent.active.count+10,quorum_thres

                    else:
```

```
                    # If registers don’t match go inactive
                    agent.active = Activity(False,0)

                    # Maybe I should suppress activity here if
                    # count reaches 0
                    #new_count = agent.active.count-1
                    #new_activity = new_count > 0
                    #agent.active = Activity(new_activity,max(new_count,0))

                    # stay active if count reaches zero
                    # agent.active = Activity(True,max(agent.active.count-1,0))

            else:

                # Agent is active, but local polled is inactive, decrement count,
                # go inactive if count is 0

                # Maybe I should suppress activity here if
                # count reaches 0
                new_count = agent.active.count-1
                new_activity = new_count > 0
                agent.active = Activity(new_activity,max(new_count,0))

        if rng.random() < error_prob:

            agent.active = Activity(agent.active.active,agent.active.count + random.cho
```

6.4 Quorum microtest

Consider making it so the other agent is passed in, not polled in this function.

This used to make agents go inactive if they polled an inactive agent in a linked/previous swarm.

```
if (
    (polled.active < quorum_threshold)
    or registers_not_right
):
    return 0
else:
    return agent.active + 1
```

but this means that any swarm polling from two swarms will go inactive half the time.

This function can do with a big cleanup, it probably should just be doing registers checking. It looks a bit complicated, but it saves me writing a ‘registers are ok’ function for each swarm.

29 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def quorum_make_test_fun29(
    swarms,
    linked_swarm_names,
    action,
    x_condition,
    rng=random,
):

    def quorum_generic_test_fun(
        linked_swarms,
        intended_action_function,
        x_condition,
        rng,
        agent,
        polled,
    ):
        next_x, next_y = intended_action_function(polled.hypothesis)

        hyp = agent.hypothesis

        # return True if registers are good. False otherwise.
        return (hyp.x == next_x and hyp.y == next_y and x_condition(hyp.x))

    linked_swarms = [
        swarms[swarm_name]
        for swarm_name
        in linked_swarm_names]

    return functools.partial(
        quorum_generic_test_fun,
        linked_swarms,
        action,
        x_condition,
        rng,)
```

Defines:
`quorum_make_test_fun`, used in chunk 31.

6.5 Quorum new hypothesis

30 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def quorum_make_hyp_fun(swarms, linked_swarm_names):

    def quorum_generic_hyp_fun(linked_swarms, rng=random):

        random_swarm = rng.choice(linked_swarms)

        polled = rng.choice(random_swarm)

        polled_x, polled_y = polled.hypothesis

        return XY(x=perturb(polled_x),y=perturb(polled_y))

    linked_swarms = [
        swarms[swarm_name]
        for swarm_name
        in linked_swarm_names]

    return functools.partial(quorum_generic_hyp_fun, linked_swarms)
```

Defines:
`quorum_make_hyp_fun`, used in chunk 31.

6.6 Quorum build new solution

31 ⟨*multi-swarm-solution.py* 10⟩+≡
```
def quorum_build_solution31(
    state_table,
    initial_x,
    initial_y,
    agent_count,
    quorum_threshold,):

    swarms = {
        name:[sds.Agent() for _ in range(agent_count)]
        for name,_,_ in state_table
    }

    for name, swarm_type, linked_swarms in state_table:

        if swarm_type == 'init':

            for agent in swarms[name]:

                agent.hypothesis = XY(x=initial_x,y=initial_y)

                agent.active = Activity(True,quorum_threshold)

        else:

            for agent in swarms[name]:

                agent.active = Activity(False,0)

                agent.hypothesis = XY(x=0,y=0)

    solution = []

    for swarm_name, swarm_type, linked_swarm_names in state_table:

        action, x_condition = swarm_types[swarm_type]

        test_fun = quorum_make_test_fun29(
            swarms,
            linked_swarm_names,
            action,
            x_condition,)

        hyp_function = quorum_make_hyp_fun30(swarms, linked_swarm_names)

        previous_swarms = [
            swarms[name]
            for name
```

```
                in linked_swarm_names]

            swarm_solution = (
                swarm_name,
                swarms[swarm_name],
                test_fun,
                hyp_function,
                previous_swarms)

            solution.append(swarm_solution)

        return solution
```

Defines:
  `quorum_build_solution`, used in chunk 36.
Uses `quorum_make_hyp_fun` 30 and `quorum_make_test_fun` 29.

6.7 Quorum iterate

At this point, ‘microtest’ is a function which takes two agents (the ‘current’ agent, and a randomly polled agent) and returns whether their registers are valid (e.g. their x registers are the same and the y register is of ‘current’ is one larger than the y register of ‘polled’) and that the ‘current’ agent has the correct value in its x register, which is either $x = 0$, $x > 0$ or ‘dont care’.

33 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def quorum_iterate33(solution, quorum_threshold, error_prob, allow_self_selection, rng=

    for swarm_name, swarm, microtest, hyp_function, previous_swarms in solution:

        quorum_diffusion_phase(
            swarm,
            allow_self_selection,
            quorum_threshold,
            hyp_function,
            error_prob,
            rng,
        )


        #for agent_num, agent in enumerate(swarm):

        #   if agent.active.active is False: # agent is inactive

        #       local_polled = random.choice(swarm)

        #       if local_polled.active.active: # polled is active or quorate

        #           agent.hypothesis = local_polled.hypothesis

        #       else:

        #           agent.hypothesis = hyp_function(random)

        #   elif agent.active.count < quorum_threshold: # agent is active

        #       local_polled = random.choice(swarm)

        #       while (not allow_self_selection) and local_polled is agent:

        #           local_polled = random.choice(swarm)

        #       if local_polled.active.active and agent.hypothesis == local_polled.hypo
        #           # polled is active and agent and polled share hypotheses

        #           # agent becomes more active capped at quorum threshold
        #           agent.active = Activity(
        #              True,
        #              min(quorum_threshold,agent.active.count+1))

        #   elif agent.active.count >= quorum_threshold: # agent is quorate

        #       # suppress a random agent from a random linked swarm
        #       random_linked_swarm = random.choice(previous_swarms)

        #       linked_polled = random.choice(random_linked_swarm)

        #       linked_polled.active = Activity(False,0)

        #Quorum test phase
        quorum_test_phase25(
            swarm,
            previous_swarms,
            quorum_threshold,
            microtest,
            error_prob,
            rng,)

    # Halting condition
    halt_swarm = next(swarm for name,swarm,_,_,_ in solution if name == 'halt')

    # Halt if all agents in the halt swarm are quorate
    return all(
        (x.active.active and x.active.count >= quorum_threshold)
        for x
        in halt_swarm
    )
```

Defines:
quorum_iterate, used in chunk 36.
Uses quorum_test_phase 25.

34 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def report_swarms34(solution):
    return tuple(
        (
            name,tuple((agent.hypothesis, agent.active)
            for agent
            in swarm)
        )
        for name,swarm,_,_,_
        in solution
    )
```

Defines:
report_swarms, used in chunk 36.

35 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def solution_to_json35(solution):
    return tuple(
        (name,
        tuple(
            (
                agent.hypothesis.x,
                agent.hypothesis.y,
                agent.active.active,
                agent.active.count,
            )
            for agent
            in swarm
        ))
        for name,swarm,_,_,_
        in solution
    )
```

Defines:
  `solution_to_json`, used in chunk 36.

6.8 Quorum execute

36 ⟨*multi-swarm-solution.py* 10⟩+≡

```
def do_quorum_sensing_solution36(
    quorum_threshold,
    initial_x,
    initial_y,
    agent_count,
    error_prob,
    allow_self_selection,
    max_iterations,
):

    if max_iterations is None:
        repeat_generator = itertools.count()
    else:
        repeat_generator = range(max_iterations)

    solution = quorum_build_solution31(
        state_table,
        initial_x=initial_x,
        initial_y=initial_y,
        agent_count=agent_count,
        quorum_threshold=quorum_threshold,)

    iteration_states = []

    for num, repeat in enumerate(repeat_generator,start=1):

        iteration_states.append(solution_to_json35(solution))

        if quorum_iterate33(
            solution,
            quorum_threshold,
            error_prob,
            allow_self_selection,
        ):

            halt_swarm = next(
                swarm
                for name,swarm,_,_,_
                in solution
                if name == 'halt')

            print('Finished at iteration {r}. Answer is {a}.'.format(
                r=repeat,
                a=next(
                    agent
                    for agent
                    in halt_swarm
                    if (
                        agent.active.active
                        and agent.active.count >= quorum_threshold
                    )
                ).hypothesis.y
            ))

            break

    else:

        print("Maximum iterations reached ({i}). No answer.".format(
            i=num
        ))

    iteration_states.append(solution_to_json$_{35}$(solution))


    print(report_swarms$_{34}$(solution))
    print(solution_to_json$_{35}$(solution))

    return iteration_states

def get_anim(x,y,quorum_threshold,max_iterations):
    raise NotImplementedError("""\
This is out of date, use a third file to make the anim then call animate.\
""")
    animation = do_quorum_sensing_solution$_{36}$(quorum_threshold, x, y,max_iterations=max_
    return animation

if __name__ == "__main__":
    animations = []

    animation_count = 10
    initial_x = 1000
    initial_y = 0
    agent_count = 10
    quorum_threshold = 100
    error_prob=0.01
    allow_self_selection = False
    max_iterations = 1000000


    for animation_num in range(animation_count):

        animation = do_quorum_sensing_solution$_{36}$(
            initial_x=initial_x,
            initial_y=initial_y,
            agent_count=agent_count,
            quorum_threshold=quorum_threshold,
            error_prob=error_prob,
            allow_self_selection=allow_self_selection,
            max_iterations=max_iterations,)

        metadata = {
            'num':animation_num,
            'x':initial_x,
            'y':initial_y,
            'agent count':agent_count,
            'quorum threshold':quorum_threshold,
            'error prob':error_prob,
            'allow self selection':allow_self_selection,
            'max iterations':max_iterations,
        }

        animations.append((metadata,animation))

        print('done animation {n}'.format(n=animation_num))

    with open('multi-swarm-animation.json','w') as f:
        for animation in animations:
            f.write(json.dumps(animation))
            f.write("\n")
        #json.dump(animations,f)
```

Defines:
  `do_quorum_sensing_solution`, never used.
Uses `quorum_build_solution` 31, `quorum_iterate` 33, `report_swarms` 34, and `solution_to_json` 35.

6.9 OLD Quorum summary pseudocode

39 ⟨*pseudo quorum sensing* 39⟩≡

```
Do until an agent in the halt swarm hits quorum

    For each swarm

        For each agent # Diffusion phase

            if agent's quorum count is 0 # polling is inactive

                poll an agent from the same swarm

                if polled agent's quorum count > 0

                    polling agent copies hypothesis from polled agent

                else

                    polling agent generates random hypothesis by polling a
                    random agent from a random linked swarm and copying the
                    hypothesis with peturbation.

        For each agent # Test phase

            poll an agent from a linked swarm

            if polling agent has not hit quorum threshold

                if polled agent has hit quorum threshold

                    if polling agent's registers are accurate

                        agent increments quorum count

                    else

                        agent sets quorum count to zero

                else

                    do nothing

            else:

                polled agent's quorum count becomes zero
```

Root chunk (not used in this document).

6.10 Quorum summary pseudocode

40 ⟨*pseudo quorum sensing 2* 40⟩≡

```
 Do until an agent in the halt swarm hits quorum

     For each agent in swarm # Diffusion phase

         if agent is inactive

             poll an agent from local swarm

             if locally polled is active or quorate

                 agent copies hypothesis from local_polled agent

             else

                 agent generates random hypothesis by copying the
                 hypothesis, with peturbation, of remotely polled agent.

         else if agent is active

             locally poll an agent from the same swarm

             if locally polled is active or quorate

                 agent becomes quorate

         else if agent is quorate

             remotely polled agent becomes inactive

     For each agent in swarm # Test phase

         if agent is quorate

             continue

         remotely poll a random agent from a random linked swarm

         if remotely polled agent is quorate

             if agent's registers are accurate

                 agent becomes active

             else

                 agent becomes inactive
```

Root chunk (not used in this document).

A Index

`build_solution`: 21, 22
`decrementX`: 18, 20b
`decrementY`: 18, 20b
`do_quorum_sensing_solution`: 36
`incrementX`: 18, 20b
`incrementY`: 18, 20b
`is_valid`: 6, 7
`make_hyp_fun`: 19, 20a, 21
`make_test_fun`: 19, 21
`noop`: 18, 19, 20b
`quorum_build_solution`: 31, 36
`quorum_iterate`: 33, 36
`quorum_make_hyp_fun`: 30, 31
`quorum_make_test_fun`: 29, 31
`quorum_test_phase`: 25, 33
`report_swarms`: 34, 36
`solution_to_json`: 35, 36
`Xdontcare`: 18, 20b
`Xeq0`: 18, 20b
`Xgt0`: 18, 19, 20b

B Code Chunks

⟨*multi-swarm-solution.py* 10⟩ 10, 11a, 11b, 12a, 12b, 12c, 13a, 13b, 14a, 14b, 15, 16, 18, 19, 20a, 20b, 20c, 21, 22, 24b, 25, 27, 29, 30, 31, 33, 34, 35, 36
⟨*pseudo quorum sensing* 39⟩ 39
⟨*pseudo quorum sensing 2* 40⟩ 40
⟨*quorum diffusion phase* 24a⟩ 24a
⟨*quorum inhibit mode* 23b⟩ 23b
⟨*quorum test phase* 23a⟩ 23a
⟨*run-two-agent-solution.py* 9⟩ 9
⟨*two-agent-solution.py* 3⟩ 3, 4, 5, 6, 7, 8