Braitenbot Code

Jake Brawer, Aaron Hill

August 31, 2015

Contents

1 This Document Or: How I Learned to Stop Worrying And Love Emacs

This document, created using Emacs' org-mode is an example of a literate program. Briefly, a literate program intersperses plain prose amongst snippets of runnable code. The result is a program that is both human-readbale and, due to its inherent modularity, easily testable.

Additionally, Emacs will automatically export the file to a Latex formated PDF or HTML file for easier human consumption. Going forward I will attempt to maintain and update this document as the main hub of the Braitenbot project.

2 What I Did This Summer

Much of the summer was spent fixing, adding to, and validating the code. Here is a small list of a few of the changes that were made:

- Decoder instructions are no longer duples, now 3-tuples, with the third element specifying the origin of the sucessive wire.
- Fixed error that resulted in threads having incomplete wire connections.
- Altered the behavior of certain methods/classes including InstructionSet, reprodue, Organism, etc (see corresponding section is this document).
- Wrote the algorithm `RankAndCrossGeneration` (and its variants) which ranks organisms in a generation based on their relative performances and crosses them accordingly.
- Other bug fixes and tweaks

A large chunk of my time was spent specifying and validating the code. I have included these in this document, but this is still a work in progress. Now that the code is in a more modular format the code can be tested more easily and therefore more robust test can be administered.

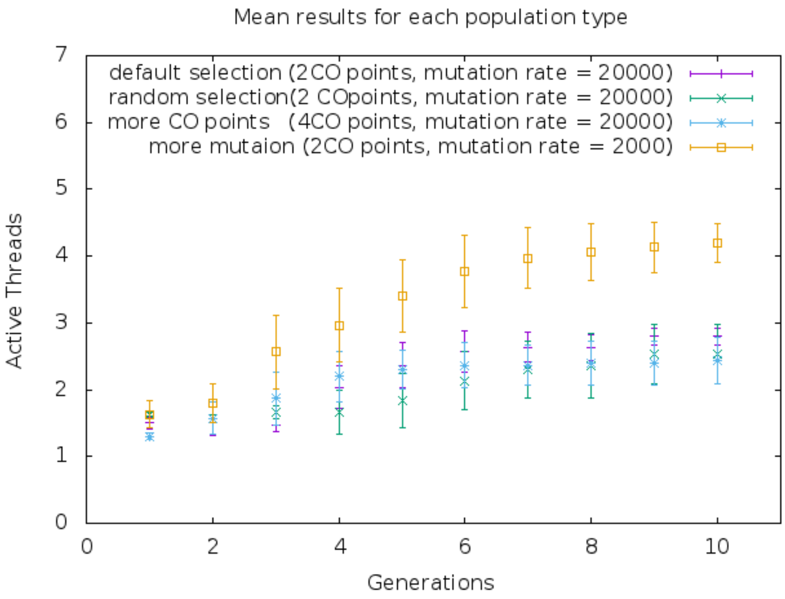
Probably most importantly, we devised a hypothesis and a corresponing experiment:

H_1: The distribution of crossover points is realted to the robustness of an organism's traits, and thus, is directly related to evolvability, fitness, and modualrity.

This could be tested by running at least two populations. Both populations would initially contain organisms with the same number of crossover points, but in one population they could be at any locus on the genome, while in the other population they would be restricted to the intergenomic regions. One might predict that the latter population would have an advantage due to potentially adaptive genes to be transmitted intact to successive generations.

2.1 What I learned

I ran many populations in simulation, selecting for organism containing the highest number of 'functional threads', i.e. threads that code for actual wires (thus, I didn't need to run actual robots. Populations differed in terms of a few key parameters, i.e. crossover rate and mutation rate. Below are the results of each paramter group (Each group representing the average of 3 populations).



Here's what I take away from these results:

- Due to many factors, including small population size,there is not much variation in the population
  - This is mitigated in part by a very high mutation rate (An order of magnitude larger than the default)
- Very high mutation rates are required because of the large amount of noncoding DNA
  - Likewise, more adding more crossover points may not affect performance significantly

due to being distributed amongst noncoding regions.

3 Next steps

The most obvious next step would be to decrease the size of the noncoding regions (by decreasing thread size) and to run these experiments again and compare the results. There experiments would also help decide which parameters to use when running the actual robot.

Additionally, before any physical expeirments are run, I would like to complete this document, including better comments and more robust validation tests.

I

also think we need to look/tweak the algorithm that generates "viable" organisms to make sure it is giving us what we want.

4 Pin and Pin Group Code

```
import random

class pin:
    # group_id represents the group the pin belongs to
    # number identifies the pin number within the group
    def __init__(self, group_id, number, group):
        self.group_id = group_id
        self.number = number
        #self.group = group
        self.available = True

    def setAvailability(self, bool):
        self.available = bool



class PinGroup(object):
    def __call__(self):
        return self

    def __init__(self):
        self.type = None

    def get_input(self, pin_index):
        raise NotImplementedError

    def get_output(self, pin_index):
        raise NotImplementedError

    def get_random_input(self):
        raise NotImplementedError
    def get_random_output(self):
        raise NotImplementedError

    def match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        pin_found = False
        for x in range(len(pin_list1)):
            if pin.group_id == pin_list1[x].group_id and pin.number == pin_list1[x].number
                pin_found = True
                # NOTE: instead of deleting the pin from the list, the pin's available var
                # this allows for the ability to determine if a pin is 'taken' by another
                pin_list1[x].setAvailability(False)
                break

        if not pin_found and pin_list2 is not None:
            for x in range(len(pin_list2)):
                # None types in the pin list signify pins that are no longer available, an
                if pin_list2[x] is not None:
                    if pin.group_id == pin_list2[x].group_id and pin.number == pin_list2[x
                        pin_found = True
                        # NOTE: instead of deleting the pin from the list, the pin's avail
                        # this allows for the ability to determine if a pin is 'taken' by
                        pin_list2[x].setAvailability(False)
                        break
        if pin_found is False:
            pass
        assert pin_found is True


    #NOTE: I(nhibitory) and E(xcitatory) are inputs
        # N and T(hreshold) are outputs




class MotorSensorPinGroup(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.pins = None

    def get_input(self, pin_index):
        target_pin = self.pins[pin_index]
        #print target_pin
        #self.match_and_remove(target_pin, self.pins)
        if target_pin.available == False:
            raise IndexError
        else:
            self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin
    def get_output(self, pin_index):
        target_pin = self.pins[pin_index]
        #self.match_and_remove(target_pin, self.pins)
        if target_pin.available ==False:
            raise IndexError
        else:
            self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def get_random_input(self):
        target_pin = random.choice([pin for pin in self.pins if pin.available is True])
        #self.match_and_remove(target_pin, self.pins)
        self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def get_random_output(self):
        target_pin = random.choice([pin for pin in self.pins if pin.available is True])
        #self.match_and_remove(target_pin, self.pins)
        self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(MotorSensorPinGroup, self).match_and_remove_pin(pin, pin_list1, pin_list2)


class Group1(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group e1
        self.e1 = [Pin("e1", i, self) for i in range(4)]
        self.i1 = [Pin("i2", i, self) for i in range(3)]
        self.n1 = [Pin("n1", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e1 + self.i1
        target_pin = all_inputs[pin_index]
        target_pin.available = False
        #self.match_and_remove(target_pin, self.e1, self.i1)
        self.call_match_and_remove_pin(target_pin, self.e1, self.i1)
        return target_pin

    def get_output(self, pin_index):
        target_pin = self.n1[pin_index]
        #self.match_and_remove(target_pin, self.n1)
        self.call_match_and_remove_pin(target_pin, self.n1)
        return target_pin

    """
    gets a random available input pin
    """
    def get_random_input(self):
        # put all available pins in a list
        available_inputs = [pin for pin in self.e1 + self.i1 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e1, self.i1)
        return target_pin

    def get_random_output(self):
        # put all available pins in a list
        target_pin = random.choice([pin for pin in self.n1 if pin.available is True])
        self.call_match_and_remove_pin(target_pin, self.n1)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group1, self).match_and_remove_pin(pin, pin_list1, pin_list2)



class Group3(PinGroup):
    def __init__(self):
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 3
        self.e3 = [Pin("e3", i, self) for i in range(4)]
        self.i3 = [Pin("i3", i, self) for i in range(3)]
        self.t3 = [Pin("t3", i, self) for i in range(4)]
        self.n3 = [Pin("n3", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e3 + self.i3
        target_pin = all_inputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.e3, self.i3)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n3 + self.t3
        target_pin = all_outputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.n3, self.t3)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e3 + self.i3 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e3, self.i3)
        return target_pin
    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n3 + self.t3 if pin.available is True]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n3, self.t3)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group3, self).match_and_remove_pin(pin, pin_list1, pin_list2)

class Group4(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 4
        self.e4 = [Pin("e4", i, self) for i in range(4)]
        self.i4 = [Pin("i4", i, self) for i in range(3)]
        self.t4 = [Pin("t4", i, self) for i in range(4)]
        self.n4 = [Pin("n4", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e4 + self.i4
        target_pin = all_inputs[pin_index]
        #self.match_and_remove(target_pin, self.e4, self.i4)
        self.call_match_and_remove_pin(target_pin, self.e4, self.i4)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n4 + self.t4
        target_pin = all_outputs[pin_index]
        #self.match_and_remove(target_pin, self.n4, self.t4)
        self.call_match_and_remove_pin(target_pin, self.n4, self.t4)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e4 + self.i4 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e4, self.i4)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n4 + self.t4 if pin.available is True]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n4, self.t4)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group4, self).match_and_remove_pin(pin, pin_list1, pin_list2)



class Group5(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 5
        self.e5 = [Pin("e5", i, self) for i in range(4)]
        self.i5 = [Pin("i5", i, self) for i in range(3)]
        self.t5 = [Pin("t5", i, self) for i in range(4)]
        self.n5 = [Pin("n5", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e5 + self.i5
        target_pin = all_inputs[pin_index]
        #self.match_and_remove(target_pin, self.e5, self.i5)
        self.call_match_and_remove_pin(target_pin, self.e5, self.i5)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n5 + self.t5
        target_pin = all_outputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.n5, self.t5)
        #self.match_and_remove(target_pin, self.n5, self.t5)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e5 + self.i5 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e5, self.i5)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n5 + self.t5 if pin.available is True]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n5, self.t5)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group5, self).match_and_remove_pin(pin, pin_list1, pin_list2)

class group6(pingroup):

    def __init__(self):
        #pingroup.__init__(self)
        super(pingroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 6
        self.e6 = [pin("e6", i, self) for i in range(4)]
        self.i6 = [pin("i6", i, self) for i in range(3)]
        self.t6 = [pin("t6", i, self) for i in range(4)]
        self.n6 = [pin("n6", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e6 + self.i6
        target_pin = all_inputs[pin_index]
        #self.match_and_remove(target_pin, self.e6, self.i6)
        self.call_match_and_remove_pin(target_pin, self.e6, self.i6)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n6 + self.t6
        target_pin = all_outputs[pin_index]
        #self.match_and_remove(target_pin, self.n6, self.t6)
        self.call_match_and_remove_pin(target_pin, self.n6, self.t6)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e6 + self.i6 if pin.available is true]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e6, self.i6)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n6 + self.t6 if pin.available is true]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n6, self.t6)
        return target_pin
    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=none):
        super(group6, self).match_and_remove_pin(pin, pin_list1, pin_list2)


class GroupPl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("pl", i, self) for i in range(6)]


class GroupRl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("rl", i, self) for i in range(6)]


class GroupRr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("rr", i, self) for i in range(6)]


class GroupPr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("pr", i, self) for i in range(6)]


class GroupBl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("bl", i, self) for i in range(4)]


class GroupBr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("br", i, self) for i in range(4)]


class GroupFl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("fl", i, self) for i in range(4)]


class GroupFr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("fr", i, self) for i in range(4)]
```

4.1 Imports

```
import random
```

4.2 Class: Pin

Input: group_{id}, a string the name of the pin group
number, an int, the pin number
Output: A Pin object with attributes group_{id}, number and availability (a bool).

```
class pin:
    # group_id represents the group the pin belongs to
    # number identifies the pin number within the group
    def __init__(self, group_id, number, group):
        self.group_id = group_id
        self.number = number
        #self.group = group
        self.available = True

    def setAvailability(self, bool):
        self.available = bool
```

4.2.1 Test:

```
class pin:
    # group_id represents the group the pin belongs to
    # number identifies the pin number within the group
    def __init__(self, group_id, number, group):
        self.group_id = group_id
        self.number = number
        #self.group = group
        self.available = True

    def setAvailability(self, bool):
        self.available = bool



p = pin(3,4,5)
print p.group_id
print p.number
```

4.3 Class: PinGroup

Input: None
Output: PinGroup object

```
class PinGroup(object):
    def __call__(self):
        return self

    def __init__(self):
        self.type = None

    def get_input(self, pin_index):
        raise NotImplementedError

    def get_output(self, pin_index):
        raise NotImplementedError

    def get_random_input(self):
        raise NotImplementedError

    def get_random_output(self):
        raise NotImplementedError

    def match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        pin_found = False
        for x in range(len(pin_list1)):
            if pin.group_id == pin_list1[x].group_id and pin.number == pin_list1[x].number
                pin_found = True
                # NOTE: instead of deleting the pin from the list, the pin's available var
                # this allows for the ability to determine if a pin is 'taken' by another 
                pin_list1[x].setAvailability(False)
                break

        if not pin_found and pin_list2 is not None:
            for x in range(len(pin_list2)):
                # None types in the pin list signify pins that are no longer available, an
                if pin_list2[x] is not None:
                    if pin.group_id == pin_list2[x].group_id and pin.number == pin_list2[x
                        pin_found = True
                        # NOTE: instead of deleting the pin from the list, the pin's avail
                        # this allows for the ability to determine if a pin is 'taken' by 
                        pin_list2[x].setAvailability(False)
                        break
        if pin_found is False:
            pass
        assert pin_found is True


    #NOTE: I(nhibitory) and E(xcitatory) are inputs
```

```
        # N and T(hreshold) are outputs
```

4.3.1 Methods

Input: pin, a Pin object $\text{pin}_{\text{list1}}$, a list containing pins $\text{pin}_{\text{list2}}$, a list containing pins **Output:** None **Side Effect:** Checks to see if pin is in either $\text{pin}_{\text{lists}}$. If so, it sets the availability of the matching pin in either list to false.

```
def match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
    pin_found = False
    for x in range(len(pin_list1)):
        if pin.group_id == pin_list1[x].group_id and pin.number == pin_list1[x].number:
            pin_found = True
            # NOTE: instead of deleting the pin from the list, the pin's available variabl
            # this allows for the ability to determine if a pin is 'taken' by another thre
            pin_list1[x].setAvailability(False)
            break

    if not pin_found and pin_list2 is not None:
        for x in range(len(pin_list2)):
            # None types in the pin list signify pins that are no longer available, and sh
            if pin_list2[x] is not None:
                if pin.group_id == pin_list2[x].group_id and pin.number == pin_list2[x].nu
                    pin_found = True
                    # NOTE: instead of deleting the pin from the list, the pin's available
                    # this allows for the ability to determine if a pin is 'taken' by anot
                    pin_list2[x].setAvailability(False)
                    break
    if pin_found is False:
        pass
    assert pin_found is True


#NOTE: I(nhibitory) and E(xcitatory) are inputs
    # N and T(hreshold) are outputs
```

1. Test

```
# Need these to test match_and_remove_pins
class pin:
    # group_id represents the group the pin belongs to
    # number identifies the pin number within the group
    def __init__(self, group_id, number, group):
        self.group_id = group_id
        self.number = number
        #self.group = group
        self.available = True

    def setAvailability(self, bool):
        self.available = bool
```

```
class PinGroup(object):
    def __call__(self):
        return self

    def __init__(self):
        self.type = None

    def get_input(self, pin_index):
        raise NotImplementedError

    def get_output(self, pin_index):
        raise NotImplementedError

    def get_random_input(self):
        raise NotImplementedError

    def get_random_output(self):
        raise NotImplementedError

    def match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        pin_found = False
        for x in range(len(pin_list1)):
            if pin.group_id == pin_list1[x].group_id and pin.number == pin_list1[x].nu
                pin_found = True
                # NOTE: instead of deleting the pin from the list, the pin's available
                # this allows for the ability to determine if a pin is 'taken' by anot
                pin_list1[x].setAvailability(False)
                break

        if not pin_found and pin_list2 is not None:
            for x in range(len(pin_list2)):
                # None types in the pin list signify pins that are no longer available
                if pin_list2[x] is not None:
                    if pin.group_id == pin_list2[x].group_id and pin.number == pin_lis
                        pin_found = True
                        # NOTE: instead of deleting the pin from the list, the pin's a
                        # this allows for the ability to determine if a pin is 'taken
                        pin_list2[x].setAvailability(False)
                        break
        if pin_found is False:
            pass
        assert pin_found is True


    #NOTE: I(nhibitory) and E(xcitatory) are inputs
        # N and T(hreshold) are outputs
pin1 = pin(2,1,1)
pin2 = pin(1,3,4)
piny = pin(1,1,1) #piny is identical to pin1, thus pin1 should be made unavailable
pingroup= PinGroup()

list1 = [pin2, pin1]
print "pin1 avaialability before match_and_remove is called:", pin1.available
pingroup.match_and_remove_pin(piny, list1)
print "Pin1 availability after:", pin1.available
```

4.4 Class: MotorSensorPinGroup

```
class MotorSensorPinGroup(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.pins = None

    def get_input(self, pin_index):
        target_pin = self.pins[pin_index]
        #print target_pin
        #self.match_and_remove(target_pin, self.pins)
        if target_pin.available == False:
            raise IndexError
        else:
            self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def get_output(self, pin_index):
        target_pin = self.pins[pin_index]
        #self.match_and_remove(target_pin, self.pins)
        if target_pin.available ==False:
            raise IndexError
        else:
            self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def get_random_input(self):
        target_pin = random.choice([pin for pin in self.pins if pin.available is True])
        #self.match_and_remove(target_pin, self.pins)
        self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def get_random_output(self):
        target_pin = random.choice([pin for pin in self.pins if pin.available is True])
        #self.match_and_remove(target_pin, self.pins)
        self.call_match_and_remove_pin(target_pin, self.pins)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(MotorSensorPinGroup, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5 Specific PinGroups

Componenets on the Braitenbot are broken up into different PinGroups.
Groups1-6 correspond to the 6 neurons

4.5.1 Class:Group1

```
class Group1(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group e1
        self.e1 = [Pin("e1", i, self) for i in range(4)]
        self.i1 = [Pin("i2", i, self) for i in range(3)]
        self.n1 = [Pin("n1", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e1 + self.i1
        target_pin = all_inputs[pin_index]
        target_pin.available = False
        #self.match_and_remove(target_pin, self.e1, self.i1)
        self.call_match_and_remove_pin(target_pin, self.e1, self.i1)
        return target_pin

    def get_output(self, pin_index):
        target_pin = self.n1[pin_index]
        #self.match_and_remove(target_pin, self.n1)
        self.call_match_and_remove_pin(target_pin, self.n1)
        return target_pin

    """
    gets a random available input pin
    """
    def get_random_input(self):
        # put all available pins in a list
        available_inputs = [pin for pin in self.e1 + self.i1 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e1, self.i1)
        return target_pin

    def get_random_output(self):
        # put all available pins in a list
        target_pin = random.choice([pin for pin in self.n1 if pin.available is True])
        self.call_match_and_remove_pin(target_pin, self.n1)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group1, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5.2 Class:Group2

#+NAME Group2

```
class Group2(PinGroup):
    def __init__(self):
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 2
        self.e2 = [Pin("e2", i, self) for i in range(4)]
        self.i2 = [Pin("i2", i, self) for i in range(3)]
        self.n2 = [Pin("n2", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e2 + self.i2
        target_pin = all_inputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.e2, self.i2)
        return target_pin

    def get_output(self, pin_index):
        target_pin = self.n2[pin_index]
        self.call_match_and_remove_pin(target_pin, self.n2)
        return target_pin

    """
    return a random available input
    """
    def get_random_input(self):
        # put all available pins in a list
        available_inputs = [pin for pin in self.e2 + self.i2 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e2, self.i2)
        return target_pin

    """
    return a random available output
    """
    def get_random_output(self):
        # put all available pins in a list
        target_pin = random.choice([pin for pin in self.n2 if pin.available is True])
        self.call_match_and_remove_pin(target_pin, self.n2)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group2, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5.3 Class:Group3

```
class Group3(PinGroup):
    def __init__(self):
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 3
        self.e3 = [Pin("e3", i, self) for i in range(4)]
        self.i3 = [Pin("i3", i, self) for i in range(3)]
        self.t3 = [Pin("t3", i, self) for i in range(4)]
        self.n3 = [Pin("n3", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e3 + self.i3
        target_pin = all_inputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.e3, self.i3)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n3 + self.t3
        target_pin = all_outputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.n3, self.t3)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e3 + self.i3 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e3, self.i3)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n3 + self.t3 if pin.available is True]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n3, self.t3)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group3, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5.4 Class:Group4

```
class Group4(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 4
        self.e4 = [Pin("e4", i, self) for i in range(4)]
        self.i4 = [Pin("i4", i, self) for i in range(3)]
        self.t4 = [Pin("t4", i, self) for i in range(4)]
        self.n4 = [Pin("n4", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e4 + self.i4
        target_pin = all_inputs[pin_index]
        #self.match_and_remove(target_pin, self.e4, self.i4)
        self.call_match_and_remove_pin(target_pin, self.e4, self.i4)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n4 + self.t4
        target_pin = all_outputs[pin_index]
        #self.match_and_remove(target_pin, self.n4, self.t4)
        self.call_match_and_remove_pin(target_pin, self.n4, self.t4)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e4 + self.i4 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e4, self.i4)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n4 + self.t4 if pin.available is True]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n4, self.t4)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group4, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5.5 Class:Group5

```
class Group5(PinGroup):
    def __init__(self):
        #PinGroup.__init__(self)
        super(PinGroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 5
        self.e5 = [Pin("e5", i, self) for i in range(4)]
        self.i5 = [Pin("i5", i, self) for i in range(3)]
        self.t5 = [Pin("t5", i, self) for i in range(4)]
        self.n5 = [Pin("n5", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e5 + self.i5
        target_pin = all_inputs[pin_index]
        #self.match_and_remove(target_pin, self.e5, self.i5)
        self.call_match_and_remove_pin(target_pin, self.e5, self.i5)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n5 + self.t5
        target_pin = all_outputs[pin_index]
        self.call_match_and_remove_pin(target_pin, self.n5, self.t5)
        #self.match_and_remove(target_pin, self.n5, self.t5)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e5 + self.i5 if pin.available is True]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e5, self.i5)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n5 + self.t5 if pin.available is True]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n5, self.t5)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=None):
        super(Group5, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5.6 Class:Group6

```
class group6(pingroup):

    def __init__(self):
        #pingroup.__init__(self)
        super(pingroup, self).__init__()
        self.type = "standard"
        # list of available pins in group 6
        self.e6 = [pin("e6", i, self) for i in range(4)]
        self.i6 = [pin("i6", i, self) for i in range(3)]
        self.t6 = [pin("t6", i, self) for i in range(4)]
        self.n6 = [pin("n6", i, self) for i in range(4)]

    def get_input(self, pin_index):
        all_inputs = self.e6 + self.i6
        target_pin = all_inputs[pin_index]
        #self.match_and_remove(target_pin, self.e6, self.i6)
        self.call_match_and_remove_pin(target_pin, self.e6, self.i6)
        return target_pin

    def get_output(self, pin_index):
        all_outputs = self.n6 + self.t6
        target_pin = all_outputs[pin_index]
        #self.match_and_remove(target_pin, self.n6, self.t6)
        self.call_match_and_remove_pin(target_pin, self.n6, self.t6)
        return target_pin

    """
    returns a random available input
    """
    def get_random_input(self):
        available_inputs = [pin for pin in self.e6 + self.i6 if pin.available is true]
        target_pin = random.choice(available_inputs)
        self.call_match_and_remove_pin(target_pin, self.e6, self.i6)
        return target_pin

    """
    returns a random available input
    """
    def get_random_output(self):
        available_outputs = [pin for pin in self.n6 + self.t6 if pin.available is true]
        target_pin = random.choice(available_outputs)
        self.call_match_and_remove_pin(target_pin, self.n6, self.t6)
        return target_pin

    def call_match_and_remove_pin(self, pin, pin_list1, pin_list2=none):
        super(group6, self).match_and_remove_pin(pin, pin_list1, pin_list2)
```

4.5.7 Classes: Motor And Sensor Groups

```
class GroupPl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("pl", i, self) for i in range(6)]


class GroupRl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("rl", i, self) for i in range(6)]


class GroupRr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("rr", i, self) for i in range(6)]


class GroupPr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("pr", i, self) for i in range(6)]


class GroupBl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("bl", i, self) for i in range(4)]


class GroupBr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("br", i, self) for i in range(4)]


class GroupFl(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("fl", i, self) for i in range(4)]


class GroupFr(MotorSensorPinGroup):
    def __init__(self):
        #MotorSensorPinGroup.__init__(self)
        super(MotorSensorPinGroup, self).__init__()
        self.pins = [Pin("fr", i, self) for i in range(4)]
```

5 Decoder Code

```
import random
import Base
```

5.1 Imports

```
import random
import Base
```

5.2 Class: Decoder

Input: None
Output: A Decoder object with the attribute index == 1.

```
class Decoder:
    def __init__(self):
        self.index = 0



    def decode_binary(self, binary_list):
        # turn binary list into a string for easy comparison
        binary_string = ""
        for binary_digit in binary_list:
            binary_string += str(binary_digit)

        # determines what number each 4bit binary string represents
        if binary_string == "0000":
            return 0
        elif binary_string == "0001":
            return 1
        elif binary_string == "0010":
            return 2
        elif binary_string == "0011":
            return 3
        elif binary_string == "0100":
            return 4
        elif binary_string == "0101":
            return 5
        elif binary_string == "0110":
            return 6
        elif binary_string == "0111":
            return 7
        elif binary_string == "1000":
            return 8
        elif binary_string == "1001":
            return 9
        elif binary_string == "1010":
            return
        elif binary_string == "1011":
            return
        elif binary_string == "1100":
            return
        elif binary_string == "1101":
            return
        elif binary_string == "1110":
            return
        elif binary_string == "1111":
            return

    def binary_to_decimal(self, binary_list):
        dec_list = []
        # step through the array in 4s as long as there are enough digits (4) to form a nu
        # this is checked through the expression (len - 4) - (5 % 4)
        #print
        for x in range(0, len(binary_list)-3, 4):
            # generate the list of binary to be decoded
            temp = [binary_list[y] for y in range(x, x+4)]
            dec_list.append(self.decode_binary(temp))
        #    print temp, dec_list
        #print
        #print 'Hypothetical # of decimal digits: %s/4 = %s. Actual #: %s'% (len(binary_li
        #print
        return dec_list
        #print


    def generate_coords(self, binaryList):
        """
        method for getting the next non-NONE value from decList
        return: either the value of decList at index self.index, unless an error is found;
        return -1
        """
        coords = []
        #this value is the height of the matrix created by the pin-group
        #HEIGHT_OF_PINGROUP = 21  -- Not sure why 21 was chosen
        HEIGHT_OF_PINGROUP = 30
        #print binaryList

        def get_next_val():
            """
            gets the next value from decList, which is the list containing the decimal tra
            If this causes an index error, -1 will be returned to avoid the error from hal
            :return: the next value form decList
            """
            #print decList
            to_return = None
            try:
                while to_return is None:
                    to_return = decList[self.index]
                    self.index += 1
            except IndexError:
                self.index = -1
                return -1
            #print "index: %s  Next decimal digit: %s" % (self.index, to_return)
            return to_return
            #print


        # the input decList must have at least one digit for the creation of the initial p
        # and 3 more for the creation of a terminal pin.
        # If this condition is met, generate initial x,y coord from first value in the arr
        decList = self.binary_to_decimal(binaryList)
        #print "Direction key: 0: y+=Distance,1:x+=distance, y+=distance, 2:x+=distance, 3
        #"4:y-=distance, 5:x-=distance, y-=distance, 6: x-=distance, 7: x-=distance, y-=di
        #print decList
        if len(decList) < 3:
            return []
        else:
            x = get_next_val()
            # the inital pin coordinate will range from zero to the length of the matrix c
            #y = random.randint(0,HEIGHT_OF_PINGROUP)
            y = get_next_val() #Jake addition: no reason we need to selcet randomly. We
                                # generate perfectly good nonrandom numbers
            #print 'Original (x,y): (%s,%s)' % (x,y)

            # append first xy coordinate in the form of a 2-tuple
           # z = get_next_val() # jake addition: this decides which pin will be the origin
                                # of the subsequent connection
            coords.append((x,y))

            # do the following for every digit after the first (since it was used to gener
            # a starting position)
            # also check for the minimum required digits for the thread instruction proces
            while self.index < len(decList):# and (len(decList) - self.index) >= 4:
                # generate the x coordinate's direction, and end pin
                # this number will be 1 through 8, corresponding to the different
                # cardinal directions
                pos1 = get_next_val()
                pos2 = get_next_val()
                pos3 = get_next_val()
                """ try:
                    pos4 = get_next_val() #Jake addition: this decides the origin
                except(IndexError):
                    pass"""
                # the pos1 and pos2 values are used for direction and cannot be negative.
                # distance, and must be greater than 0
                if pos1 < 0 or pos2 < 0 or pos3 <= 0:
                    #print 'Break! a decimal <= 0 was generated'
                    #print 'possible culprits: pos1:%s,pos2:%s,pos3:%s' % (pos1,pos2,pos3)
                    break

                direction = (pos1 + pos2) % 8
                distance = pos3
```

```
                if direction == 0:
                    y += distance
                elif direction == 1:
                    y += distance
                    x += distance
                elif direction == 2:
                    x += distance
                elif direction == 3:
                    x += distance
                    y -= distance
                elif direction == 4:
                    y -= distance
                elif direction == 5:
                    y -= distance
                    x -= distance
                elif direction == 6:
                    x -= distance
                elif direction == 7:
                    y += distance
                    x -= distance
                if x < 0 or y < 0:
                    #print 'Break! x or y < 0'
                    #print '(%s,%s)' % (x,y)
                    break
                #print'Direction: (next_val + next_val ) mod 8 --> (%s + %s) mod 8 = %s' %
                #print 'Distance: next_val ---> %s' % distance
                #print 'Direction: %s, Distance: %s --->(%s,%s)' % (direction, distance ,x
               # if self.index in [5 +i*3 for i in range(len(decList))]:
               #Jake addition: adds third coordiante, z :which determines origin
               #of the subsequent wire connection in a thread.
                z = get_next_val()
                if z < 0:
                    #print 'Break! z < 0'
                    #print'z = %s' % z
                    break
                else:
                     coords.append((x,y,z))
                #print
                #print 'Coord z: %s. Final coords: (%s,%s,%s)' % (z,x,y,z)
               # else:
                #    coords.append((x, y))

        #print 'Resultant Coords:', coords
        self.index = 0
        return coords
```

5.2.1 Methods

1. decode binary **Input:** binary$_{\text{list}}$, a 4-bit list **Output:** The corresponding decimal digits for numbers 0-9 only.

```
def decode_binary(self, binary_list):
    # turn binary list into a string for easy comparison
    binary_string = ""
    for binary_digit in binary_list:
        binary_string += str(binary_digit)

    # determines what number each 4bit binary string represents
    if binary_string == "0000":
        return 0
    elif binary_string == "0001":
        return 1
    elif binary_string == "0010":
        return 2
    elif binary_string == "0011":
        return 3
    elif binary_string == "0100":
        return 4
    elif binary_string == "0101":
        return 5
    elif binary_string == "0110":
        return 6
    elif binary_string == "0111":
        return 7
    elif binary_string == "1000":
        return 8
    elif binary_string == "1001":
        return 9
    elif binary_string == "1010":
        return
    elif binary_string == "1011":
        return
    elif binary_string == "1100":
        return
    elif binary_string == "1101":
        return
    elif binary_string == "1110":
        return
    elif binary_string == "1111":
        return
```

2. binary to decimal **Input:** $\text{binary}_{\text{list}}$, an n-bit list **Output:** A list containing the corresponding decimal digits between 0-9 only. **Process:** Appends decimal digits to a list calculated by inputting every 4 digits of binary-list into $\text{decode}_{\text{binary}}$.

```
def binary_to_decimal(self, binary_list):
    dec_list = []
    # step through the array in 4s as long as there are enough digits (4) to form a nu
    # this is checked through the expression (len - 4) - (5 % 4)
    #print
    for x in range(0, len(binary_list)-3, 4):
        # generate the list of binary to be decoded
        temp = [binary_list[y] for y in range(x, x+4)]
        dec_list.append(self.decode_binary(temp))
    #    print temp, dec_list
    #print
    #print 'Hypothetical # of decimal digits: %s/4 = %s. Actual #: %s'% (len(binary_li
    #print
    return dec_list
    #print
```

3. generate coords

   Input: binaryList, an n-bit list **Output:** A list of 2- and 3-tuples in the form x,y and x,y,z respectively where the first tuple in the list is a 2-tuple, and the rest are 3-tuples. A given tuples values are dependent upon the values contained within the preceding tuple, in a process outlined more in depth below. **Side Effect:** decList, a list of decimal and none values used by other methods.

```
def generate_coords(self, binaryList):
    """
    method for getting the next non-NONE value from decList
    return: either the value of decList at index self.index, unless an error is found
    return -1
    """
    coords = []
    #this value is the height of the matrix created by the pin-group
    #HEIGHT_OF_PINGROUP = 21  -- Not sure why 21 was chosen
    HEIGHT_OF_PINGROUP = 30
    #print binaryList

    def get_next_val():
        """
        gets the next value from decList, which is the list containing the decimal tra
        If this causes an index error, -1 will be returned to avoid the error from hal
        :return: the next value form decList
        """
        #print decList
        to_return = None
        try:
            while to_return is None:
                to_return = decList[self.index]
                self.index += 1
        except IndexError:
            self.index = -1
            return -1
        #print "index: %s  Next decimal digit: %s" % (self.index, to_return)
        return to_return
        #print
```

```
# the input decList must have at least one digit for the creation of the initial p
# and 3 more for the creation of a terminal pin.
# If this condition is met, generate initial x,y coord from first value in the ar
decList = self.binary_to_decimal(binaryList)
#print "Direction key: 0: y+=Distance,1:x+=distance, y+=distance, 2:x+=distance, 
#"4:y-=distance, 5:x-=distance, y-=distance, 6: x-=distance, 7: x-=distance, y-=di
#print decList
if len(decList) < 3:
    return []
else:
    x = get_next_val()
    # the inital pin coordinate will range from zero to the length of the matrix 
    #y = random.randint(0,HEIGHT_OF_PINGROUP)
    y = get_next_val() #Jake addition: no reason we need to selcet randomly. We
                        # generate perfectly good nonrandom numbers
    #print ’Original (x,y): (%s,%s)’ % (x,y)

    # append first xy coordinate in the form of a 2-tuple
   # z = get_next_val() # jake addition: this decides which pin will be the origi
                        # of the subsequent connection
    coords.append((x,y))

    # do the following for every digit after the first (since it was used to gene
    # a starting position)
    # also check for the minimum required digits for the thread instruction proces
    while self.index < len(decList):# and (len(decList) - self.index) >= 4:
        # generate the x coordinate’s direction, and end pin
        # this number will be 1 through 8, corresponding to the different
        # cardinal directions
        pos1 = get_next_val()
        pos2 = get_next_val()
        pos3 = get_next_val()
        """ try:
            pos4 = get_next_val() #Jake addition: this decides the origin
        except(IndexError):
            pass"""
        # the pos1 and pos2 values are used for direction and cannot be negative.
        # distance, and must be greater than 0
        if pos1 < 0 or pos2 < 0 or pos3 <= 0:
            #print ’Break! a decimal <= 0 was generated’
            #print ’possible culprits: pos1:%s,pos2:%s,pos3:%s’ % (pos1,pos2,pos3
            break

        direction = (pos1 + pos2) % 8
        distance = pos3
        if direction == 0:
            y += distance
        elif direction == 1:
            y += distance
            x += distance
        elif direction == 2:
            x += distance
        elif direction == 3:
            x += distance
            y -= distance
        elif direction == 4:
            y -= distance
        elif direction == 5:
            y -= distance
            x -= distance
        elif direction == 6:
            x -= distance
        elif direction == 7:
            y += distance
            x -= distance
        if x < 0 or y < 0:
            #print 'Break! x or y < 0'
            #print '(%s,%s)' % (x,y)
            break
        #print'Direction: (next_val + next_val ) mod 8 --> (%s + %s) mod 8 = %s' %
        #print 'Distance: next_val ---> %s' % distance
        #print 'Direction: %s, Distance: %s --->(%s,%s)' % (direction, distance ,x
       # if self.index in [5 +i*3 for i in range(len(decList))]:
       #Jake addition: adds third coordiante, z :which determines origin
       #of the subsequent wire connection in a thread.
        z = get_next_val()
        if z < 0:
            #print 'Break! z < 0'
            #print'z = %s' % z
            break
        else:
             coords.append((x,y,z))
        #print
        #print 'Coord z: %s. Final coords: (%s,%s,%s)' % (z,x,y,z)
       # else:
        #    coords.append((x, y))

#print 'Resultant Coords:', coords
self.index = 0
return coords
```

(a) Submethod: get next val
Input: decList, A list of decimal and None values. **Output:** Returns the next non-None value in decList, or -1 if an IndexError is raised.

```
def get_next_val():
    """
    gets the next value from decList, which is the list containing the decimal tr
    If this causes an index error, -1 will be returned to avoid the error from ha
    :return: the next value form decList
    """
    #print decList
    to_return = None
    try:
        while to_return is None:
            to_return = decList[self.index]
            self.index += 1
    except IndexError:
        self.index = -1
        return -1
    #print "index: %s  Next decimal digit: %s" % (self.index, to_return)
    return to_return
    #print


# the input decList must have at least one digit for the creation of the initial
# and 3 more for the creation of a terminal pin.
# If this condition is met, generate initial x,y coord from first value in the ar
decList = self.binary_to_decimal(binaryList)
#print "Direction key: 0: y+=Distance,1:x+=distance, y+=distance, 2:x+=distance,
#"4:y-=distance, 5:x-=distance, y-=distance, 6: x-=distance, 7: x-=distance, y-=d
#print decList
if len(decList) < 3:
    return []
else:
    x = get_next_val()
    # the inital pin coordinate will range from zero to the length of the matrix
    #y = random.randint(0,HEIGHT_OF_PINGROUP)
    y = get_next_val() #Jake addition: no reason we need to selcet randomly. We
                        # generate perfectly good nonrandom numbers
    #print 'Original (x,y): (%s,%s)' % (x,y)

    # append first xy coordinate in the form of a 2-tuple
   # z = get_next_val() # jake addition: this decides which pin will be the origi
                        # of the subsequent connection
    coords.append((x,y))

    # do the following for every digit after the first (since it was used to gene
    # a starting position)
    # also check for the minimum required digits for the thread instruction proce
    while self.index < len(decList):# and (len(decList) - self.index) >= 4:
        # generate the x coordinate's direction, and end pin
        # this number will be 1 through 8, corresponding to the different
        # cardinal directions
        pos1 = get_next_val()
        pos2 = get_next_val()
        pos3 = get_next_val()
        """ try:
            pos4 = get_next_val() #Jake addition: this decides the origin
        except(IndexError):
```

```
     pass"""
 # the pos1 and pos2 values are used for direction and cannot be negative.
 # distance, and must be greater than 0
 if pos1 < 0 or pos2 < 0 or pos3 <= 0:
     #print ’Break! a decimal <= 0 was generated’
     #print ’possible culprits: pos1:%s,pos2:%s,pos3:%s’ % (pos1,pos2,pos3
     break

 direction = (pos1 + pos2) % 8
 distance = pos3
 if direction == 0:
     y += distance
 elif direction == 1:
     y += distance
     x += distance
 elif direction == 2:
     x += distance
 elif direction == 3:
     x += distance
     y -= distance
 elif direction == 4:
     y -= distance
 elif direction == 5:
     y -= distance
     x -= distance
 elif direction == 6:
     x -= distance
 elif direction == 7:
     y += distance
     x -= distance
 if x < 0 or y < 0:
     #print ’Break! x or y < 0’
     #print ’(%s,%s)’ % (x,y)
     break
 #print’Direction: (next_val + next_val ) mod 8 --> (%s + %s) mod 8 = %s’
 #print ’Distance: next_val ---> %s’ % distance
 #print ’Direction: %s, Distance: %s --->(%s,%s)’ % (direction, distance ,
# if self.index in [5 +i*3 for i in range(len(decList))]:
#Jake addition: adds third coordiante, z :which determines origin
#of the subsequent wire connection in a thread.
 z = get_next_val()
 if z < 0:
     #print ’Break! z < 0’
     #print’z = %s’ % z
     break
 else:
      coords.append((x,y,z))
 #print
 #print ’Coord z: %s. Final coords: (%s,%s,%s)’ % (z,x,y,z)
# else:
```

```
                 #    coords.append((x, y))

         #print 'Resultant Coords:', coords
         self.index = 0
         return coords
```

6 Base And InstrutionSet Code

```
import random
import string


class Base:
    def __init__(self):
        self.char = random.randint(0, 1)
        self.crossover_point = 0 # Crossover hotspots are set later by InstructionSet

    def set_crossover_point(self, new_val):
            self.crossover_point = new_val
            return self.crossover_point

    def set_char(self, new_val):
            self.char = new_val
            return self.char


class InstructionSet:
    def __init__(self, size, crossover_point_number,unrestricted_distribution, gene_length
        self.genome = []
        x = size  # a place holder, the length of the genome
        counter = 0
        for num in range(0, x ):
            self.genome.append(Base())
            # in the event there are no break points at all
            # maybe we dont want this though? Can discuss later
        if unrestricted_distribution:
            while counter != crossover_point_number:
                random.choice(self.genome).set_crossover_point(1)
                counter +=1
        else:
            potential_locations = [i*gene_length for i in range (1, (len(self.genome)/gene
            while counter != crossover_point_number:
                rand_index = random.choice(potential_locations)
                self.genome[rand_index].set_crossover_point(1)
                potential_locations.remove(rand_index)
                counter +=1
            print potential_locations
        assert counter == crossover_point_number
        """for s in self.genome:
            counter += s.crossover_point
        if counter < 1:
            random.choice(self.genome).set_crossover_point(1)"""

    def setGenome(self, new_genome):
        self.genome = new_genome


    def mutate(self):
        #mutation_chance = 20000 #THIS IS THE REAL ONE
        mutation_chance = 20000
        for i in range(len(self.genome)):
            rand_int1 = random.randint(1, mutation_chance)
            rand_int2 = random.randint(1, mutation_chance)
            if rand_int1 == mutation_chance:
                print 'Crossover_point mutation at index: %s' % i
                if self.genome[i].crossover_point == 0:
                    self.genome[i].set_crossover_point(1)
                    print '0 --> %s' % self.genome[i].crossover_point
                    return True
                else:
                    self.genome[i].set_crossover_point(0)
                    print '1 --> %s' % self.genome[i].crossover_point
                    return True
            if rand_int2 == mutation_chance:
                print 'Char mutation at index: %s' % i
                if self.genome[i].char == 0:
                    self.genome[i].set_char(1)
                    print '0 --> %s' % self.genome[i].char
                    return True
                else:
                    self.genome[i].set_char(0)
                    print '1 --> %s' % self.genome[i].char
```

6.1 Imports

```
import random
import string
```

6.2 Class: Base

input: None
Output: A Base object with two binary attributes, char and $\text{crossover}_{\text{point}}$.Char has 1/2 chance of being 1 or 0, $\text{crossover}_{\text{point}}$ is initialized to 0.

```
class Base:
    def __init__(self):
        self.char = random.randint(0, 1)
        self.crossover_point = 0 # Crossover hotspots are set later by InstructionSet
    def set_crossover_point(self, new_val):
            self.crossover_point = new_val
            return self.crossover_point

    def set_char(self, new_val):
            self.char = new_val
            return self.char
```

6.3 Class: InstructionSet

Input: None
Output: An InstructionSet object with a genome attribute. A genome is a list containing 2000 Base objects of which at least one has a crossover$_{\text{point}}$ value == 1.

```
class InstructionSet:
    def __init__(self, size, crossover_point_number,unrestricted_distribution, gene_length
        self.genome = []
        x = size  # a place holder, the length of the genome
        counter = 0
        for num in range(0, x ):
            self.genome.append(Base())
            # in the event there are no break points at all
            # maybe we dont want this though? Can discuss later
        if unrestricted_distribution:
            while counter != crossover_point_number:
                random.choice(self.genome).set_crossover_point(1)
                counter +=1
        else:
            potential_locations = [i*gene_length for i in range (1, (len(self.genome)/gene
            while counter != crossover_point_number:
                rand_index = random.choice(potential_locations)
                self.genome[rand_index].set_crossover_point(1)
                potential_locations.remove(rand_index)
                counter +=1
            print potential_locations
        assert counter == crossover_point_number
        """for s in self.genome:
            counter += s.crossover_point
        if counter < 1:
            random.choice(self.genome).set_crossover_point(1)"""

    def setGenome(self, new_genome):
        self.genome = new_genome


    def mutate(self):
        #mutation_chance = 20000 #THIS IS THE REAL ONE
        mutation_chance = 20000
        for i in range(len(self.genome)):
            rand_int1 = random.randint(1, mutation_chance)
            rand_int2 = random.randint(1, mutation_chance)
            if rand_int1 == mutation_chance:
                print ’Crossover_point mutation at index: %s’ % i
                if self.genome[i].crossover_point == 0:
                    self.genome[i].set_crossover_point(1)
                    print ’0 --> %s’ % self.genome[i].crossover_point
                    return True
                else:
                    self.genome[i].set_crossover_point(0)
                    print ’1 --> %s’ % self.genome[i].crossover_point
                    return True
            if rand_int2 == mutation_chance:
                print ’Char mutation at index: %s’ % i
                if self.genome[i].char == 0:
                    self.genome[i].set_char(1)
                    print ’0 --> %s’ % self.genome[i].char
                    return True
                else:
                    self.genome[i].set_char(0)
                    print ’1 --> %s’ % self.genome[i].char
```

6.3.1 Validation

Validating that the various intended properities of an InstructionSet hold

```
import random
import string

class Base:
    def __init__(self):
        self.char = random.randint(0, 1)
        self.crossover_point = 0 # Crossover hotspots are set later by InstructionSet

    def set_crossover_point(self, new_val):
            self.crossover_point = new_val
            return self.crossover_point

    def set_char(self, new_val):
            self.char = new_val
            return self.char

class InstructionSet:
    def __init__(self, size, crossover_point_number,unrestricted_distribution, gene_length
        self.genome = []
        x = size  # a place holder, the length of the genome
        counter = 0
        for num in range(0, x ):
```

```
        self.genome.append(Base())
        # in the event there are no break points at all
        # maybe we dont want this though? Can discuss later
    if unrestricted_distribution:
        while counter != crossover_point_number:
            random.choice(self.genome).set_crossover_point(1)
            counter +=1
    else:
        potential_locations = [i*gene_length for i in range (1, (len(self.genome)/gene
        while counter != crossover_point_number:
            rand_index = random.choice(potential_locations)
            self.genome[rand_index].set_crossover_point(1)
            potential_locations.remove(rand_index)
            counter +=1
        print potential_locations
    assert counter == crossover_point_number
    """for s in self.genome:
        counter += s.crossover_point
    if counter < 1:
        random.choice(self.genome).set_crossover_point(1)"""

def setGenome(self, new_genome):
    self.genome = new_genome


def mutate(self):
    #mutation_chance = 20000 #THIS IS THE REAL ONE
    mutation_chance = 20000
    for i in range(len(self.genome)):
        rand_int1 = random.randint(1, mutation_chance)
        rand_int2 = random.randint(1, mutation_chance)
        if rand_int1 == mutation_chance:
            print ’Crossover_point mutation at index: %s’ % i
            if self.genome[i].crossover_point == 0:
                self.genome[i].set_crossover_point(1)
                print ’0 --> %s’ % self.genome[i].crossover_point
                return True
            else:
                self.genome[i].set_crossover_point(0)
                print ’1 --> %s’ % self.genome[i].crossover_point
                return True
        if rand_int2 == mutation_chance:
            print ’Char mutation at index: %s’ % i
            if self.genome[i].char == 0:
                self.genome[i].set_char(1)
                print ’0 --> %s’ % self.genome[i].char
                return True
            else:
                self.genome[i].set_char(0)
                print ’1 --> %s’ % self.genome[i].char
```

```
def instruction_set_test(val,size, crossover_point_num, distro, gene_length):
    print '%s InstructionSets generated, each should have %s crossover points:' % (val, cr
    while val > 0:
        crossover_ps  = 0
        genome = InstructionSet(size, crossover_point_num,distro, gene_length)
        length = len(genome.genome)
        for i in range (len(genome.genome)):
            #print g.char,
            if genome.genome[i].crossover_point == 1:
                print '\nCO_point at index: %s' % i
                crossover_ps += 1
        print
        print 'InstructionSet %s length: %s, # of Crossover_points: %s' % (11 -val, length
        print
        val -= 1


instruction_set_test(10, 20,2, True, 5)
```

6.3.2 Method: mutate

Input: Nothing
Output: None
Side Effect:*Potentially modifies some of the Bases in an InstructionSets genome (char and $\text{crossover}_{\text{point}}$ values)
***Process:** The algorithm walks through each Base in an InstructionSets genome. For each Base attribute a random int between 0 and $\text{mutation}_{\text{chance}}$ is generated. If the random int == $\text{mutation}_{\text{chance}}$, the value of that attribute is changed.

```
def mutate(self):
    #mutation_chance = 20000 #THIS IS THE REAL ONE
    mutation_chance = 20000
    for i in range(len(self.genome)):
        rand_int1 = random.randint(1, mutation_chance)
        rand_int2 = random.randint(1, mutation_chance)
        if rand_int1 == mutation_chance:
            print 'Crossover_point mutation at index: %s' % i
            if self.genome[i].crossover_point == 0:
                self.genome[i].set_crossover_point(1)
                print '0 --> %s' % self.genome[i].crossover_point
                return True
            else:
                self.genome[i].set_crossover_point(0)
                print '1 --> %s' % self.genome[i].crossover_point
                return True
```

```
if rand_int2 == mutation_chance:
    print 'Char mutation at index: %s' % i
    if self.genome[i].char == 0:
        self.genome[i].set_char(1)
        print '0 --> %s' % self.genome[i].char
        return True
    else:
        self.genome[i].set_char(0)
        print '1 --> %s' % self.genome[i].char
```

1. Validation Vaidatinf that the function mutate mutates and InstructonSet as many times as expected

```
import random
import string

class Base:
    def __init__(self):
        self.char = random.randint(0, 1)
        self.crossover_point = 0 # Crossover hotspots are set later by InstructionSet

    def set_crossover_point(self, new_val):
            self.crossover_point = new_val
            return self.crossover_point

    def set_char(self, new_val):
            self.char = new_val
            return self.char

class InstructionSet:
    def __init__(self, size, crossover_point_number,unrestricted_distribution, gene_le
        self.genome = []
        x = size  # a place holder, the length of the genome
        counter = 0
        for num in range(0, x ):
            self.genome.append(Base())
            # in the event there are no break points at all
            # maybe we dont want this though? Can discuss later
        if unrestricted_distribution:
            while counter != crossover_point_number:
                random.choice(self.genome).set_crossover_point(1)
                counter +=1
        else:
            potential_locations = [i*gene_length for i in range (1, (len(self.genome)/
            while counter != crossover_point_number:
                rand_index = random.choice(potential_locations)
                self.genome[rand_index].set_crossover_point(1)
                potential_locations.remove(rand_index)
                counter +=1
            print potential_locations
        assert counter == crossover_point_number
        """for s in self.genome:
            counter += s.crossover_point
        if counter < 1:
            random.choice(self.genome).set_crossover_point(1)"""

    def setGenome(self, new_genome):
        self.genome = new_genome


    def mutate(self):
        #mutation_chance = 20000 #THIS IS THE REAL ONE
        mutation_chance = 20000
        for i in range(len(self.genome)):
            rand_int1 = random.randint(1, mutation_chance)
            rand_int2 = random.randint(1, mutation_chance)
            if rand_int1 == mutation_chance:
                print 'Crossover_point mutation at index: %s' % i
                if self.genome[i].crossover_point == 0:
                    self.genome[i].set_crossover_point(1)
                    print '0 --> %s' % self.genome[i].crossover_point
                    return True
                else:
                    self.genome[i].set_crossover_point(0)
                    print '1 --> %s' % self.genome[i].crossover_point
                    return True
            if rand_int2 == mutation_chance:
                print 'Char mutation at index: %s' % i
                if self.genome[i].char == 0:
                    self.genome[i].set_char(1)
                    print '0 --> %s' % self.genome[i].char
                    return True
                else:
                    self.genome[i].set_char(0)
                    print '1 --> %s' % self.genome[i].char


def mutation_test(val):
    print 'Results of running mutate %s times ' % val
    genome = InstructionSet(2000, 2, True, 20)
    count = 0
    for i in range (0, val):
        if genome.mutate():
            count += 1
    print 'For each Base in InstructionSet, there is 2/20000 of the Base being mutatec

mutation_test(100)
```

7 Thread And Organism Code

```
class Thread:
    def __init__(self, thread_decoder):
        self.binary = []
        self.decoded_instructions = []
        self.connected_pins = []
        self.decoder = thread_decoder

    # simply calls the decoder to decode the thread’s instructions
    def decode(self):
        self.decoded_instructions = self.decoder.generate_coords(self.binary)
class Organism:
    def __init__(self, generation, generational_index,genome_size, num_crossover_points, u
        # store perfromance on behavioral task
        self.performance_1 = None
        self.performance_2 = None
        self.reproduction_possibilities = None
        self.generation = generation
        self.generational_index = generational_index
        # store organizational and naming information
        #NOTE: no longer saves a reference to parent org object
        #as that resulted in gigundus file sizes
        #try-except block necessary because parents may be None
        try:
            self.parent1_generation = parent1.generation
            self.parent1_generational_index = parent1.generational_index
            self.parent2_generation = parent2.generation
            self.parent2_generational_index = parent2.generational_index
        except AttributeError:
            pass
        self.filename = self.set_file_name()
        thread_length = thread_length
        self.instruction_set = InstructionSet(genome_size, num_crossover_points,unrestrict
        #This conditional is recquired for threads to build with
        # recombinated genome
        if genome is None: self.genome = self.instruction_set.genome
        else: self.genome = genome
        self.decoder = Decoder()
        # initialize pin groups
        self.group1 = Group1()
        self.group2 = Group2()
        self.group3 = Group3()
        self.group4 = Group4()
        self.group5 = Group5()
        self.group6 = Group6()
        self.groupPl = GroupPl()
        self.groupRl = GroupRl()
        self.groupRr = GroupRr()
        self.groupPr = GroupPr()
        self.groupBl = GroupBl()
        self.groupBr = GroupBr()
        self.groupFl = GroupFl()
        self.groupFr = GroupFr()
        # organize pin groups into a single list
        self.pinGroups = [self.group1, self.group2, self.group3, self.group4, self.group5,
                          self.groupRl, self.groupRr, self.groupPr, self.groupBl, self.gro
        # threads will eventually be created and appended to the thread list
        self.threads = []
        # store the pins currently connected in the organism (in no specific order)
        self.connections = []

        self.create_threads(thread_length)
        self.generate_thread_instructions()
        self.build_thread_coordinates()
```

7.1 Imports

```
from BaseAndInstructionSet import *
from Decoder import Decoder
from PinAndPingroup import *
import random
import os
import jsonpickle
```

7.2 Class: Thread

Input: thread$_{\text{decoder}}$, a Decoder object
Output: a Thread, stores a section of an Organism's InstructionSet and builds connections from it, whcih are also stored.

```
class Thread:
    def __init__(self, thread_decoder):
        self.binary = []
        self.decoded_instructions = []
        self.connected_pins = []
        self.decoder = thread_decoder

    # simply calls the decoder to decode the thread's instructions
    def decode(self):
        self.decoded_instructions = self.decoder.generate_coords(self.binary)
```

7.3 Class: Organism

Input: generation, int, the generation the org belongs to.
generational index, int, tracks the order in which the orgs in a gen were created
parent1=None, Organism, One of the orgs parents, defaults to None
parent2=None, Organism, The other parent, also defaults to none

genome=None: An InstructionSet, defaults to None.

Output: An Organism object. It keeps track of an individual's genome, lineage, and experimental performance, as well as builds its phenotype from the genome.

```
class Organism:
    def __init__(self, generation, generational_index,genome_size, num_crossover_points, u
        # store perfromance on behavioral task
        self.performance_1 = None
        self.performance_2 = None
        self.reproduction_possibilities = None
        self.generation = generation
        self.generational_index = generational_index
        # store organizational and naming information
        #NOTE: no longer saves a reference to parent org object
        #as that resulted in gigundus file sizes
        #try-except block necessary because parents may be None
        try:
            self.parent1_generation = parent1.generation
            self.parent1_generational_index = parent1.generational_index
            self.parent2_generation = parent2.generation
            self.parent2_generational_index = parent2.generational_index
        except AttributeError:
            pass
        self.filename = self.set_file_name()
        thread_length = thread_length
        self.instruction_set = InstructionSet(genome_size, num_crossover_points,unrestrict
        #This conditional is recquired for threads to build with
        # recombinated genome
        if genome is None: self.genome = self.instruction_set.genome
        else: self.genome = genome
        self.decoder = Decoder()
        # initialize pin groups
        self.group1 = Group1()
        self.group2 = Group2()
        self.group3 = Group3()
        self.group4 = Group4()
        self.group5 = Group5()
        self.group6 = Group6()
        self.groupPl = GroupPl()
        self.groupRl = GroupRl()
        self.groupRr = GroupRr()
        self.groupPr = GroupPr()
        self.groupBl = GroupBl()
        self.groupBr = GroupBr()
        self.groupFl = GroupFl()
        self.groupFr = GroupFr()
        # organize pin groups into a single list
        self.pinGroups = [self.group1, self.group2, self.group3, self.group4, self.group5,
                          self.groupRl, self.groupRr, self.groupPr, self.groupBl, self.gro
        # threads will eventually be created and appended to the thread list
        self.threads = []
        # store the pins currently connected in the organism (in no specific order)
        self.connections = []

        self.create_threads(thread_length)
        self.generate_thread_instructions()
        self.build_thread_coordinates()
```

7.3.1 Class Methods

```
def save_to_file(self, path):
    dir = os.mkdir(path+"/"+self.filename)
    with open(path+"/"+self.filename+"/"+self.filename+".txt", 'wb') as output:
        data = jsonpickle.encode(self)
        output.write(data)
def create_threads(self, thread_length):
    for genome_index in range(0, len(self.genome), thread_length):
        # iteratively create lists of base chars of size 'thread_length'
        # these lists will become the binary for the threads
        new_thread = Thread(self.decoder)
        try:
            # get the chars from each base in the segment of the instruction code being ex
            thread_binary = ([self.genome[i].char for i in range(genome_index, \ genome_in
            new_thread.binary = thread_binary
            self.threads.append(new_thread)
        # in the event of not having enough bases to create an entire thread
        # let the thread be truncated, and stop copying over bases, and append it to the l
        except IndexError:
            thread_binary = ([self.genome[i].char for i in range(genome_index, len(self.ge
            new_thread.binary = thread_binary
            self.threads.append(new_thread)
def generate_thread_instructions(self):
    for thread in self.threads:
        # instructions are xy coordinate points to plug into the pinGroups
        thread.decode()
        #print thread.decoded_instructions

<build_thread_coordinates>>
def is_viable(self):
    connected_pins = []

    def check1():
        for connected_pin_group in connected_pins:
            if (#("bl" in connected_pin_group and "fr" in connected_pin_group) or
                   # ("fl" in connected_pin_group and "br" in connected_pin_group) or
                    ("bl" in connected_pin_group and "br" in connected_pin_group ) or
                    ("fl" in connected_pin_group and "fr" in connected_pin_group)):
                return True
        return False
```

```
def check3():
    for connected_pin_group in connected_pins:
        if ((#"rr" in connected_pin_group or
                     #"rl" in connected_pin_group or
                     "pl" in connected_pin_group or
                     "pr" in connected_pin_group) and
                ("fl" in connected_pin_group or
                         "bl" in connected_pin_group or
                         "fr" in connected_pin_group or
                         "br" in connected_pin_group)):

            return True
        return False

def check4():
    try:
         if connected_pins[0] ==connected_pins[1] and connected_pins\
            [len(connected_pins) - 1]\
                  ==  connected_pins[len(connected_pins) - 2]:
                False
         else:
                True
    except(IndexError):
        pass

for t in self.threads:
    if len(t.connected_pins) > 0:
        # make a set out of the connected pins of the thread
        t_set = set([pin.group_id for pin in t.connected_pins])
        connected_pins.append(t_set)
        # loop through the list, and for every group of connected pins, check the \
            #intersection of it &
        # and its neighbor.
        # If there is an intersection, place the union of the two sets in the connecte
        # group and remove the two original sets. This will determine if the correct p
        # to create a viable phenotype
        for x in range(len(connected_pins)-1):
            if len(set(connected_pins[x]).intersection(set(connected_pins[x+1]))) > 0:
                merged_set = set(connected_pins[x]).union(connected_pins[x+1])
                connected_pins.remove(connected_pins[x+1])
                connected_pins.remove(connected_pins[x])
                connected_pins.append(merged_set)
                # check to see if the length of the connected_pin set has changed due\
                    #to appends and removes
                if x < len(connected_pins)-1:
                    break


if check1() and check3( ):  # and check2():
    #print "connected pins: ", connected_pins
    return True
else:
    return False
```

1. set file name **Input:** None
   Output: A unique string for identifying a particular organism, containing generational info as well as the name of the Organism's parents.
   #+NAME; set$_{\text{filename}}$

   ```
   """
   creates the string for the organism's filename
   """
   def set_file_name(self):
       #if self.parent1 is not None and self.parent2 is not None:
       try:
           filename = (str(self.generation) + "_" +
                       str(self.generational_index) + "_" +
                       str(self.parent1_generation) + "_" +
                       str(self.parent1_generational_index) + "_" +
                       str(self.parent2_generation) + "_" +
                       str(self.parent2_generational_index))
       except AttributeError:
           filename = (str(self.generation) + "_" +
                       str(self.generational_index) + "_" +
                       str(" ") + "_" +
                       str(" ") + "_" +
                       str(" ") + "_" +
                       str(" "))
       return filename
   ```

2. save to file **Input:** path: full path to desired location
   Output: a new directory named after the Organism, containing a pickled instantiation of the Organism.

   ```
   def save_to_file(self, path):
       dir = os.mkdir(path+"/"+self.filename)
       with open(path+"/"+self.filename+"/"+self.filename+".txt", 'wb') as output:
           data = jsonpickle.encode(self)
           output.write(data)
   ```

3. create threads

   ```
   def create_threads(self, thread_length):
       for genome_index in range(0, len(self.genome), thread_length):
           # iteratively create lists of base chars of size 'thread_length'
           # these lists will become the binary for the threads
           new_thread = Thread(self.decoder)
           try:
   ```

```
                # get the chars from each base in the segment of the instruction code bei
                thread_binary = ([self.genome[i].char for i in range(genome_index, \ genom
                new_thread.binary = thread_binary
                self.threads.append(new_thread)
            # in the event of not having enough bases to create an entire thread
            # let the thread be truncated, and stop copying over bases, and append it to t
            except IndexError:
                thread_binary = ([self.genome[i].char for i in range(genome_index, len(sel
                new_thread.binary = thread_binary
                self.threads.append(new_thread)
```

4. generate thread instructions **Input:** Nothing
   Output: Nothing
   Side Effect: The binary instructions for each Thread in self.threads (see above) is decoded into corresponding coordinate instructions (see Decoder).

```
def generate_thread_instructions(self):
    for thread in self.threads:
        # instructions are xy coordinate points to plug into the pinGroups
        thread.decode()
        #print thread.decoded_instructions
```

5. build thread coordinates **Input:** Nothing
   Output: Nothing
   Side Effect: Determines the pins connected as dictated by the coordinates of each thread.
   Process: Each Thread is 'built' (i.e. their decoded$_{\text{instructions}}$ are used to accesses PinGroups and Pins (see below)) using a round-robin approach. This done by simultaneously building each thread, one index at a time. Threads that are actively being built are stored in the list active$_{\text{threads}}$. Threads are removed from active$_{\text{threads}}$ if they collided with with a previously built Thread, for trying to accesses out of bounds Pins, for having only one valid pin, etc. Pins are accessed using the xyz coordinates stored in Thread.decoded$_{\text{instructions}}$, where x corresponds to the PinGroup, y corresponds to a specific Pin in the PinGroup, and z corresponding to another Pin within that same PinGroup– the origin of the next wire. After each Thread is built, and therefore active$_{\text{threads}}$ is empty, threads are checked to make sure there are no connections without a terminal pin.

```
def build_thread_coordinates(self):
    # threads will be temporarily copied into a separate list of running threads, to c
    # making their connections is completed
    running_threads = []
    for thread in self.threads:
        # we only want to use the the threads that connect at least two pins.
        # this is represented by the number of instructions in said thread
        if len(thread.decoded_instructions) >= 2:
```

```
        running_threads.append(thread)

# using a round-robin approach attempt to pair a thread's coordinate to a pin. whe
# some reason (i.e. collision between threads, or coordinates not corresponding to
# the thread will not be runnable and be taken from the running_threads list
index = 0
#tracks which threads have been run, and in turn, when the index should be increme
num_threads_run = 0
active_threads = [i for i in running_threads] #A deepcopy that we are free to modi
while len(active_threads) > 0:
    #print '\nThread index: %s' % index
    for running in running_threads:
        # check the next index in all of running thread when all threads have been
        if num_threads_run % len(running_threads):#len(running_threads):
            index += 1
            #print "---------------------------------------------\nNew Index:  %s"
            num_threads_run = 0

        error_encountered = False
        # declare variables for finding and storing a selected pin
        if running in active_threads:
            #print '\nActive Thread Coords:', running.decoded_instructions
            try:
                # get the specific pin coordinates from the instruction and transl
                pin_coordinates = running.decoded_instructions[index]
                accessed_pin_group = self.pinGroups[pin_coordinates[0]]
                accessed_output_pin = accessed_pin_group.get_input(pin_coordinates
                #print "Coords: %s  Group : %s  Pin: %s" % (pin_coordinates, acces
                # Jake addition 2015-06-09 this hopefully chooses another pin to b
                # ofrthe next connection (same pin group as terminus of previous c
            # print pin_coordinates,

            # an index error means that the thread's coordinates could not connect
            except IndexError:
                try:
                    #print "Out of Bounds coordinate: %s. Thread deactivated" %  s
                    pass
                except IndexError:
                    pass
                #print 'Bad index: %s' % index
                error_encountered = True
                # if a thread only has one pin, then it cannot create a connection
                if len(running.connected_pins) == 1:
                    to_remove = running.connected_pins[0]
                    # set the pin's availability to 'true'
                    to_remove.available = True
                    # remove the pin from the thread's & organism's group of conne
                    for x in range(len(self.connections)):
                        if (self.connections[x].group_id == to_remove.group_id and
                                    self.connections[x].number == to_remove.number
                            del self.connections[x]
                            break
                    # wipe the running thread’s connected pins since it only conta
                    running.connected_pins = []
                active_threads.remove(running)

        # it is possible that the pin exists but has been taken
            if not error_encountered:
                try:
                    # ensure the pin hasn’t been ’taken’ by another thread alread
                    if accessed_output_pin in self.connections:
                        #print "pin already taken: %s" % accessed_output_pin.grou
                        raise LookupError("Connection failed: pin already connect
                    ###WARNING: OUTDATED CODE
                    # its possible the accessed pin is unavailable, signifying it
                    #if not accessed_pin.available:
                    #    raise LookupError("Connection failed: pin already connec
                    else:
                        self.connections.append(accessed_output_pin)
                        running.connected_pins.append(accessed_output_pin)

                    #print ’connected pins:’,[i.group_id for i in running.connect
                    if len(pin_coordinates) == 3: #and (len(running.decoded_instr
                        new_connection_origin = accessed_pin_group.get_output(pin_
                    else:
                        new_connection_origin = None
                        # ensure the pin hasn’t been ’taken’ by another thread al
                        # connect to a random input pin in the same group
                        # input pins are used since the previous pin was an outpu
                        #output_pin = accessed_pin_group.get_random_input()
                        #self.connections.append(output_pin)
                        #running.connected_pins.append(output_pin)
                    if new_connection_origin is not None:
                        if new_connection_origin in self.connections:
                            raise LookupError("Connection failed: pinalready conn
                        else:
                            self.connections.append(new_connection_origin)
                            running.connected_pins.append(new_connection_origin)

                except LookupError:
                    # if a thread only has two pins, then it cannot create a conn
                    # group, and each pin must be made available
                    if len(running.connected_pins) == 2:
                        error_encountered = True
                        for x in range(len(running.connected_pins)):
                            # set the pin’s availability to ’true’
                            running.connected_pins[x].available = True
                            # remove the pin from the thread’s & organism’s group
                            #self.connections.remove(running.connected_pins[x])
                            for n in range(len(self.connections)):
```

```
                                    if (self.connections[n].group_id == running.conne
                                        self.connections[n].number == running.connect
                                        del self.connections[n]
                                        break

                            # wipe the running thread's connected pins since it only c
                            # which is not a complete connection
                            running.connected_pins = []
                        active_threads.remove(running)
                        if len(running.connected_pins) >  2:
                                pass
        num_threads_run += 1

    for running in self.threads:
        if len(running.connected_pins) % 2 != 0:# and \
                #len(running.connected_pins) >= 1:
            x =len(running.connected_pins)- 1
            to_remove =  running.connected_pins[-1]
            to_remove.available = True
            running.connected_pins.remove(to_remove)
            #running.connected_pins[len(running.connected_pins) - 1].available = True
            connections_copy = [n for n in self.connections] #deepcopy that we can ma
                                                            #with impunity
            for n in self.connections:
                if (n.group_id == to_remove.group_id and\
                    n.number == to_remove.number):
                    connections_copy.remove(n)
            self.connections = connections_copy
            #running.connected_pins = [running.connected_pins[i] for i in range(x - 1
            #print 'thread stuff \n' +  [i.group_id for i in running.connected_pins]
        else:
            #for running in running_threads:
            pass
```

6. is viable **Input:** Nothing
   Output: Boolean depending on whether there is a sensorimotor connection present in an Organism's phenotype. **Process:** instantiates s ˆ m C, where s sensory PinGroup, m motor PinGroup and C is the set of all connected pins in a given thread.

```
def is_viable(self):
    connected_pins = []

    def check1():
        for connected_pin_group in connected_pins:
            if (#("bl" in connected_pin_group and "fr" in connected_pin_group) or
                   # ("fl" in connected_pin_group and "br" in connected_pin_group) or
                    ("bl" in connected_pin_group and "br" in connected_pin_group ) or
                    ("fl" in connected_pin_group and "fr" in connected_pin_group)):
```

```
            return True
    return False

def check3():
    for connected_pin_group in connected_pins:
        if ((#"rr" in connected_pin_group or
                     #"rl" in connected_pin_group or
                     "pl" in connected_pin_group or
                     "pr" in connected_pin_group) and
                ("fl" in connected_pin_group or
                         "bl" in connected_pin_group or
                         "fr" in connected_pin_group or
                         "br" in connected_pin_group)):

            return True
        return False

def check4():
    try:
         if connected_pins[0] ==connected_pins[1] and connected_pins\
            [len(connected_pins) - 1]\
                  ==  connected_pins[len(connected_pins) - 2]:
                False
         else:
                True
    except(IndexError):
        pass

for t in self.threads:
    if len(t.connected_pins) > 0:
        # make a set out of the connected pins of the thread
        t_set = set([pin.group_id for pin in t.connected_pins])
        connected_pins.append(t_set)
        # loop through the list, and for every group of connected pins, check the
            #intersection of it &
        # and its neighbor.
        # If there is an intersection, place the union of the two sets in the conn
        # group and remove the two original sets. This will determine if the corre
        # to create a viable phenotype
        for x in range(len(connected_pins)-1):
            if len(set(connected_pins[x]).intersection(set(connected_pins[x+1])))
                merged_set = set(connected_pins[x]).union(connected_pins[x+1])
                connected_pins.remove(connected_pins[x+1])
                connected_pins.remove(connected_pins[x])
                connected_pins.append(merged_set)
                # check to see if the length of the connected_pin set has changed
                    #to appends and removes
                if x < len(connected_pins)-1:
                    break
```

```
if check1() and check3( ):  # and check2():
    #print "connected pins: ", connected_pins
    return True
else:
    return False
```

7.3.2 Other Methods

1. Method: reproduce **Input:** org1: an Organism
   org2: an Organism
   path: path to the directory where the offspring will be saved. **Output:** An Organism with a recombinant genome from org1 and org2's genetic material, and saved (via pickle) in a directory located at path.
   Process: A parent is chosen at random to be the 'dominant' and 'recessive' parent. The algorithm first starts copying the Bases from the dominant's InstructionSet to child1$_{\text{genome}}$. When it reaches a Base with a crossover$_{\text{point}}$ value equal to 1, it begins copying Bases starting from the successive locus in recessive parent's InstructionSet. This switch will occur every time a crossover$_{\text{point}}$ value of 1 is encountered. A new Organism is then instantiated with the resultant recombinant genome, and is saved to a new directory (bearing its name) located at path.

```
def reproduce(org1, org2, path):
    dom = random.choice([org1, org2])  # Parent whose crossover points are being used
    rec = filter(lambda y: y != dom, [org1, org2])
    rec = rec[0]# Other parent
    child1_genome = []
    gen_count = 0
    index = 0
    # This is how the offsprings genome is made
    #allows for crossing over at nonhotspots at 1/100000 chance.
    """"while index < len(dom.genome):
        child1_genome.append(dom.genome[index])
        if dom.genome[index].crossover_point == 1:
            while dom.genome[index + 1].crossover_point != 1 and \
                    index + 1 < len(dom.genome) - 1:
                        child1_genome.append(rec.genome[index + 1])
                        index += 1
        index += 1"""

    dom_genome_copy = True
    dom_stuff =[]
    rec_stuff=[]
    while index <= len(dom.genome) - 1:
        """if index  % 4 == 0:
            dom_stuff.append('')
            #rec_stuff.append('|')"""
        if dom_genome_copy:
            child1_genome.append(dom.genome[index])
            dom_stuff.append(dom.genome[index].char)
            rec_stuff.append(rec.genome[index].char)
            if dom.genome[index].crossover_point == 1:
                dom_stuff.append('HERE')
                rec_stuff.append('HERE')
                dom_genome_copy = False
            index += 1
        else:
            child1_genome.append(rec.genome[index])
            dom_stuff.append(rec.genome[index].char)
            rec_stuff.append(rec.genome[index].char)
            if rec.genome[index].crossover_point == 1:
                dom_stuff.append('HERE')
                rec_stuff.append('HERE')
                dom_genome_copy = True
            index +=1
    """"for i in range (0, len(dom_stuff)- 1):
        print '%s  %s' %  (dom_stuff[i], rec_stuff[i])
    print dom_stuff"""


    # This takes care of  of saving the Org.
    # if the path specified does not exist a new directory
    # will be created

    count = 0
    if os.path.isdir(path):
        for root, dirs, files in os.walk(path, topdown=False):
            for name in files:
                count += 1
        child_instruction_set = InstructionSet(2100, 2,True,300)
        child_instruction_set.setGenome(child1_genome)
        child_instruction_set.mutate()
        child1 = Organism(dom.generation + 1, count,2100,2,True,300, dom, rec, child_i
    else:
        os.makedirs(path)
        child_instruction_set = InstructionSet(2100, 2,True,300)
        child_instruction_set.setGenome(child1_genome)
        child_instruction_set.mutate()
        child1 = Organism(dom.generation + 1, 0,2100,2,True,300, dom, rec, child_instr
        #print [i.char for i in child1.genome]
   # print 'child %s threads:' % child1.filename
   # for thread in child1.threads:
   #     print thread.decoded_instructions
   #     print [i.group_id for i in thread.connected_pins]
    child1.save_to_file(path)
   # print 'Dom  Rec  Crossover  real_offspring'
   # for i in range(len(child1_genome) - 1):
   #     print '%s       %s      %s          %s' % (dom.genome[i].char, rec.genome[i].
    #if is_same_genome(dom, child1): print ’THEYRE SAME’
    #else: print ’THYRE DIFF’
    return child1
```

2. Method: generate viable Generates a viable organism

```
def generate_viable():
    # writes a ’progress bar’ to the console
    def progress(x):
        out = ’\r %s organisms tested’ % x  # The output
        print out,

    genomes_tested = 0
    finished = False
    while not finished:
        test = Organism(0, 0)
        if test.is_viable():
            print "------------------------------------//"
            print "connections: "
            for thread in test.threads:
                print "new thread connections:"
                for connection in thread.connected_pins:
                    print connection.group_id, connection.number
            print "------------------------------------//"
            finished = True
        else:
            del test
            genomes_tested += 1
            progress(genomes_tested)
```