# SDS Library Documentation

Andrew Owen Martin
a.martin@gold.ac.uk
Goldsmiths, University of London
Department of Computing

December 18, 2017

## Contents

First we define an agent, then the functions for the Test Phase, the Diffusion Phase, Halting functions, Iteration functions...

# 1 Agent

An agent maintains two variables. `activity` and `hypothesis`. The `Agent` class also has a method for initialising a swarm of a given size, all agents are initialised as inactive and with no hypothesis. This necessitates that the Diffusion Phase is run first, such that all agents generate a hypothesis before the test phase.

Defining `__slots__` ensures agents can have no attributes other than those listed, for a slight reduction in memory usage.

4 ⟨*agent class definition* 4⟩≡ (80)

```
class Agent:
    """\
Data structure for defining an SDS Agent, can only maintain the
attributes 'hypothesis' and 'active'.\
"""

    __slots__ = ('hypothesis','active')

    def __init__(self, hypothesis=None, active=False):

        self.hypothesis = hypothesis

        self.active = active

    ⟨initialise a list of agents function 5a⟩

    def __iter__(self):
        yield self.hypothesis
        yield self.active
```

Defines:
`Agent`, used in chunks 5a, 30, 46, 47, 50a, 61, 62, 64, 65, 67, 73, and 75.

## 1.1 Initialisation

5a ⟨*initialise a list of agents function* 5a⟩≡ (4)

```
@staticmethod
def initialise(agent_count):
    """\
Returns a list of length agent_count, of inactive Agents with no
hypothesis; suitable for use as a swarm. For example::
    swarm = sds.Agent.initialise(agent_count=1000)
"""
    return [
        Agent(hypothesis=None, active=False)
        for _
        in range(agent_count)
    ]
```

Defines:
`initialise`, used in chunks 46, 47, 61, 62, 64, 65, 67, 73, and 75.
Uses `Agent` 4.

Here are two ways to initialise a swarm of 100 agents. Firstly, with the `initialise` function.

```
swarm = Agent.initialise(agent_count=100)
```

Secondly, with a list comprehension.

```
agent_count=100
swarm = [Agent() for _ in range(agent_count)]
```

## 1.2 Agent tuple

It may prove useful to have a minimal implementation of a read only and hashable object for an agent.

5b ⟨*agent namedtuple* 5b⟩≡ (80)

```
ReadOnlyAgent = namedtuple("ReadOnlyAgent",("hypothesis","active"))

ReadOnlyAgent.__doc__ = """\
namedtuple representation of an agent.
Attributes are hypothesis and active.\
"""
```

5c ⟨*library dependencies* 5c⟩≡ (80) 32 ▷

```
from collections import namedtuple
```

You can therefore make a read-only copy of a swarm like this.

```
    read_only_swarm = [ReadOnlyAgent._make(agent) for agent in swarm]
```

This works because `Agent` is iterable and `collections.namedtuple._make` maps a sequence into the attributes of a `namedtuple`.

# 2 Test phase

## 2.1 Generic test phase

6 ⟨*generic test phase function* 6⟩≡ (80)

```
def generic_test_phase(
    swarm,
    microtests,
    multitesting=1,
    multitest_function=all,
    scalar=False,
    synchronous=True,
    rng=random,
):
    """\
Perform a test phase. Fully configurable. Consider using the more
convenient and readable functions test_phase and scalar_test_phase.\
"""

    def make_test(hyp):
        return multitest_function(
            rng.choice(microtests)(hyp)
            for test_num
            in range(multitesting)
        )

    if scalar and synchronous:

        test_results = (
            make_test(agent.hypothesis)
            for agent
            in swarm
        )

        test_results = list(test_results)

        test_results = [
            test_result > rng.choice(test_results)
            for test_result
            in test_results
        ]

        for agent, test_result in zip(swarm, test_results):

            agent.active = test_result
            yield

    else:
```

```
        for agent in swarm:

            generic_single_agent_test(
                agent,
                swarm,
                microtests,
                scalar,
                multitesting,
                multitest_function,
            )

            yield
```

Defines:
`generic_test_phase`, used in chunks 8, 9, and 73.
Uses `generic_single_agent_test` 10, `scalar_test_phase` 9, and `test_phase` 8.

## 2.2 Standard test phase

This is a destructive function, modifying the swarm in place.

8 ⟨*test phase function* 8⟩≡ (80)

```
def test_phase(
    swarm,
    microtests,
    multitesting=1,
    multitest_function=lambda scores: next(x for x in scores),
    synchronous=True,
    rng=random,
):
    """\
Perform a test phase with boolean microtests.

This function returns a generator which must be consumed once for each
agent.\
"""

    test_phase_generator = generic_test_phase(
        swarm=swarm,
        microtests=microtests,
        multitesting=multitesting,
        multitest_function=multitest_function,
        scalar=False,
        synchronous=synchronous,
        rng=rng,
    )

    for _ in test_phase_generator:
        yield
```

Defines:
`test_phase`, used in chunks 6, 9, 23, 25, 27, 30, 34, and 62.
Uses `generic_test_phase` 6.

## 2.3 Scalar test phase

This test phase variant determines the activity of each agent by comparing the score of their test functions with that of a randomly chosen agent.

9 ⟨*scalar test phase function* 9⟩≡ (80)

```
def scalar_test_phase(
    swarm,
    microtests,
    multitesting=1,
    multitest_function=max,
    synchronous=True,
    rng=random,
):
    """\
Performs a test phase with scalar microtests, an agent becomes active
if their microtest result is larger than that of a randomly chosen
agent.

This function returns a generator which must be consumed once for each
agent.\
"""

    test_phase = generic_test_phase(
        swarm=swarm,
        microtests=microtests,
        multitesting=multitesting,
        multitest_function=multitest_function,
        scalar=True,
        synchronous=synchronous,
        rng=rng,
    )

    for _ in test_phase:
        yield
```

Defines:
`scalar_test_phase`, used in chunks 6 and 27.
Uses `generic_test_phase` 6 and `test_phase` 8.

## 2.4 Generic single agent test

Note that in the scalar version, potentially both agents perform a different microtest (or set of microtests if multitesting $> 1$).

10 ⟨*generic single agent test function* 10⟩≡ (80)

```
def generic_single_agent_test(
    agent,
    swarm,
    microtests,
    scalar,
    multitesting,
    multitest_function,
    rng=random,
):
    """\
Peform a random microtest, and set the activity, for a single agent.\
"""

    test_result = multitest_function(
        rng.choice(microtests)(agent.hypothesis)
        for multitest_num
        in range(multitesting)
    )

    if scalar:

        polled_hypothesis = rng.choice(swarm).hypothesis

        polled_result = multitest_function(
            rng.choice(microtests)(polled_hypothesis)
            for multitest_num
            in range(multitesting)
        )

        test_result = test_result > polled_result

    agent.active = test_result
```

Defines:
`generic_single_agent_test`, used in chunks 6 and 29.
Uses `activity` 45.

# 3 Diffusion phase

These are all destructive functions, modifying the swarm in place.

## 3.1 Active? Dual?

Can they be done? Bish wants them to be switches separate from context-sensitive and context-free.

## 3.2 Passive diffusion

This is the diffusion function used in Standard SDS.

11 ⟨*passive diffusion function* 11⟩≡ (80)

```
def passive_diffusion(
    swarm,
    random_hypothesis_function,
    multidiffusion=1,
    rng=random,
):
    """\
Perform a passive diffusion phase.

This function returns a generator which must be consumed once for each
agent.\
"""

    diffusion_phase = generic_diffusion(
        swarm,
        random_hypothesis_function,
        context_free=False,
        context_sensitive=False,
        multidiffusion=multidiffusion,
        rng=rng,
    )

    for _ in diffusion_phase:
        yield
```

Defines:
`passive_diffusion`, used in chunks 16, 30, 50b, 53b, 61, 62, 64, 65, 67, 69, 73, and 75.
Uses `generic_diffusion` 16.

### 3.3 Context-free diffusion

Similar to passive diffusion, only active agents may become inactive and generate a new hypothesis if they poll an agent which is also active. As active agents may be modified by this process it is necessary to take a snapshot of the state of the swarm before any modification happens, this is the `old_swarm` variable which is populated at the beginning of the phase and never modified.

12 ⟨*context free diffusion function* 12⟩≡ (80)

```
def context_free_diffusion(
    swarm,
    random_hypothesis_function,
    multidiffusion=1,
    rng=random,
):
    """\
Perform a context free diffusion phase.

This function returns a generator which must be consumed once for each
agent.\
"""

    diffusion_phase = generic_diffusion(
        swarm,
        random_hypothesis_function,
        context_free=True,
        context_sensitive=False,
        multidiffusion=multidiffusion,
        rng=rng,
    )

    for _ in diffusion_phase:
        yield
```

Defines:
`context_free_diffusion`, used in chunks 16, 27, 51, and 69.
Uses `generic_diffusion` 16.

### 3.4 Context-sensitive diffusion

Similar to context-free diffusion, only active agents may become inactive and generate a new hypothesis if they poll an agent which is active and both agents share the same hypothesis.

13 ⟨*context sensitive diffusion function* 13⟩≡ (80)

```
def context_sensitive_diffusion(
    swarm,
    random_hypothesis_function,
    multidiffusion=1,
    rng=random,
):
    """\
Perform a context sensitive diffusion phase.

This function returns a generator which must be consumed once for each
agent.\
"""

    diffusion_phase = generic_diffusion(
        swarm,
        random_hypothesis_function,
        context_free=True,
        context_sensitive=True,
        multidiffusion=multidiffusion,
        rng=rng,
    )

    for _ in diffusion_phase:
        yield
```

Defines:
`context_sensitive_diffusion`, used in chunks 16, 27, 52, 65, 69, and 73.
Uses `generic_diffusion` 16.

### 3.5 Generic diffusion

The behaviour of passive, context-free and context-sensitive diffusion can be captured in a single, five-variable truth table (Table 1), and then represented as a five-variable Karnaugh map (Table 2). From Table 2 one can derive the logic for Algorithm 1.

| CF (E) | CS (D) | a.active (A) | p.active (B) | hyp==hyp (C) | Response |
|---|---|---|---|---|---|
| F | F | F | F | F | Random hyp |
| F | F | F | F | T | Random hyp |
| F | F | F | T | F | Copy hyp |
| F | F | F | T | T | Copy hyp |
| F | F | T | F | F | Maintain hyp |
| F | F | T | F | T | Maintain hyp |
| F | F | T | T | F | Maintain hyp |
| F | F | T | T | T | Maintain hyp |
| F | T | F | F | F | Don't care |
| F | T | F | F | T | Don't care |
| F | T | F | T | F | Don't care |
| F | T | F | T | T | Don't care |
| F | T | T | F | F | Don't care |
| F | T | T | F | T | Don't care |
| F | T | T | T | F | Don't care |
| F | T | T | T | T | Don't care |
| T | F | F | F | F | Random hyp |
| T | F | F | F | T | Random hyp |
| T | F | F | T | F | Copy hyp |
| T | F | F | T | T | Copy hyp |
| T | F | T | F | F | Maintain hyp |
| T | F | T | F | T | Maintain hyp |
| T | F | T | T | F | Random hyp |
| T | F | T | T | T | Random hyp |
| T | T | F | F | F | Random hyp |
| T | T | F | F | T | Random hyp |
| T | T | F | T | F | Copy hyp |
| T | T | F | T | T | Copy hyp |
| T | T | T | F | F | Maintain hyp |
| T | T | T | F | T | Maintain hyp |
| T | T | T | T | F | Maintain hyp |
| T | T | T | T | T | Random hyp |

Table 1: The truth table for a combination of passive, context-free and context-sensitive diffusion. CF=isContextFree, CS=isContextSensitive, a.active=agentIsActive, p.active=polledIsActive, hyp==hyp=hypEqualsHyp

**Algorithm 1** Generic diffusion

**if** $\overline{A} \wedge B$ **then**
    Copy hypothesis (State 2)
**else if** $\overline{A} \vee B \wedge E \wedge (C \vee \overline{D})$ **then**
    Random hypothesis (State 1)
**else**
    Maintain hypothesis (State 3)

| | | $\overline{C}$ $\overline{D}$ $\overline{E}$ | $\overline{C}$ $\overline{D}$ $E$ | $\overline{C}$ $D$ $E$ | $\overline{C}$ $D$ $\overline{E}$ | $C$ $\overline{D}$ $\overline{E}$ | $C$ $\overline{D}$ $E$ | $C$ $D$ $E$ | $C$ $D$ $\overline{E}$ |
|---|---|---|---|---|---|---|---|---|---|
| $\overline{A}$ | $\overline{B}$ | 1 | 1 | 1 | X | 1 | 1 | 1 | X |
| $\overline{A}$ | $B$ | 2 | 2 | 2 | X | 2 | 2 | 2 | X |
| $A$ | $B$ | 3 | 1 | 3 | X | 3 | 1 | 1 | X |
| $A$ | $\overline{B}$ | 3 | 3 | 3 | X | 3 | 3 | 3 | X |

Table 2: Five variable Karnaugh map of the diffusion truth table (Table 1). A = agent is active, B = polled agent is active, C = agent and polled agents share a hypothesis, D = context-sensitive diffusion, E = context-free diffusion. 1 is random hypothesis, 2 is copy hypothesis, 3 is maintain hypothesis.

16 ⟨*generic diffusion function* 16⟩≡ (80)

```
def generic_diffusion(
    swarm,
    random_hypothesis_function,
    context_free,
    context_sensitive,
    multidiffusion,
    rng=random,
):
    """\
Perform a diffusion phase, fully configurable. Consider using the more
convenient and readable functions passive_diffusion,
context_free_diffusion and context_sensitive_diffusion.

This function returns a generator which must be consumed once for each
agent.\
"""

    if context_sensitive:
        context_free = True

    if context_free:
        old_swarm = [ReadOnlyAgent(a.hypothesis,a.active) for a in swarm]
    else:
        old_swarm = swarm

    for agent in swarm:

        generic_single_agent_diffusion(
            agent,
            old_swarm,
            random_hypothesis_function,
            context_free,
            context_sensitive,
            multidiffusion,
            rng,
        )

        yield
```

Defines:
`generic_diffusion`, used in chunks 11–13.

Uses `context_free_diffusion` 12, `context_sensitive_diffusion` 13, `generic_single_agent_diffusion` 17, and `passive_diffusion` 11.

## 3.6 Generic single agent diffusion

17 ⟨*generic single agent diffusion function* 17⟩≡ (80)

```
def generic_single_agent_diffusion(
    agent,
    swarm,
    random_hypothesis_function,
    context_free,
    context_sensitive,
    multidiffusion,
    rng=random,
):
    """\
Perform diffusion, and set the hypothesis for a single agent.\
"""
    ⟨handle multidiffusion 18⟩

    if (not agent.active and polled_agent.active):
        agent.hypothesis = polled_agent.hypothesis
    elif (
        not agent.active
        or polled_agent.active
        and context_free
        and (
            not context_sensitive
            or agent.hypothesis == polled_agent.hypothesis
        )
    ):
        agent.active = False
        agent.hypothesis = random_hypothesis_function(random)
```

Defines:
`generic_single_agent_diffusion`, used in chunks 16 and 29.

### 3.7 Multidiffusion

The job here is to populate polled_agent appropriately, and the way it is done is by setting polled_agent to the elements of a random agent generator and stopping when an active agent is reached. If an active agent is never reached then the diffusion will have failed. The remainder variable allows for a “real” number of diffusions, where a fraction of a diffusion is a randomly polled agent, followed by a chance of receiving a known inactive agent in its place.

18 ⟨*handle multidiffusion* 18⟩≡ (17)

```
polled_agents = (
    random.choice(swarm)
    for diffusion_num
    in range(int(multidiffusion))
)

for polled_agent in polled_agents:
    if polled_agent.active:
        break
else:
    remainder = multidiffusion - int(multidiffusion)

    if remainder > 0:

        polled_agent = random.choice(swarm)

        if random.random() > remainder:

            polled_agent = ReadOnlyAgent(None,False)
```

### 3.7.1 Probabilistic Rounding Function

The multidiffusion and multitesting functions may benefit from using this probabilistic rounding function. 7.1 gets rounded to 8 with probability 0.1, and rounded to 7 with probability 0.9.

19a ⟨*probabilistic rounding function* 19a⟩≡

```
def probabilistic_round(num):
    """\
Probabilistically round a number, the remainder is the probability of
rounding up.\
"""

    full = int(num)
    remainder = num - full
    if remainder > 0:
        roundup = random.random() < remainder
    return full + roundup
```

Defines:
`probabilistic_round`, never used.

# 4 Halting functions

## 4.1 Never halt function

19b ⟨*never halt function* 19b⟩≡ (80)

```
def never_halt(*args, **kwargs):
    """\
Always returns false, suitable as a halting function for a perpetual
SDS.\
"""

    return False
```

Defines:
`never_halt`, used in chunks 30, 34, 62, 67, and 69.

## 4.2 Stability halt function

20 ⟨*stability halt function* 20⟩≡ (80)

```
def make_stability_halting_function(lower, region, time):
    """\
Returns a function suitable for use as a halting function. Halts, by
returning True when it detects stability, defined by:

lower: Lower bound of proportion of activity of stability window.

region: Amount above lower which defines the upper bound of the
stability window.

time: Number of consecutive times this function must be called with
arguments within the stability window before it will halt.\
"""

    def generator_front_end(activity_count, halt_generator):
        next(halt_generator)
        halted = halt_generator.send(activity_count)
        return halted

    def is_stable_generator(lower, region, time):

        success_count = 0

        while True:

            swarm = yield

            active_count = sum(1 for agent in swarm if agent.active)/len(swarm)

            if active_count < lower or active_count > lower + region:
                success_count = 0
            else:
                success_count += 1

            yield success_count >= time

    halting_generator = is_stable_generator(lower, region, time)

    return functools.partial(
        generator_front_end,
        halt_generator=halting_generator,)
```

Defines:
`make_stability_halting_function`, used in chunk 69.
Uses `activity` 45.

## 4.3 Instant threshold halt function

21 ⟨*instant threshold halt function* 21⟩≡ (80)

```
def make_instant_threshold_halt_function(threshold):
    """\
Returns a function suitable for use as a halting function. Halts, by
returning True when the proportion of global activity is greater than
threshold.\
"""
    def threshold_halt_function(swarm, threshold):

        activity = sum(1 for agent in swarm if agent.active)/len(swarm)

        return activity > threshold

    return functools.partial(threshold_halt_function,threshold=threshold)
```

Defines:
`make_instant_threshold_halt_function`, used in chunk 69.
Uses `activity` 45.

## 4.4 Threshold time halt function

22 ⟨*threshold time halt function* 22⟩≡ (80)

```
def make_threshold_time_halting_function(lower, time):
    """\
Returns a function suitable for use as a halting function. Halts, by
returning True when the proportion of global activity is greater than
threshold for a number of calls to this function defined by time.\
"""

    def generator_front_end(activity_count, halt_generator):
        next(halt_generator)
        halted = halt_generator.send(activity_count)
        return halted

    def is_stable_generator(lower, time):

        success_count = 0

        while True:

            swarm = yield

            active_count = sum(1 for agent in swarm if agent.active)/len(swarm)

            if active_count < lower:
                success_count = 0
            else:
                success_count += 1

            yield success_count >= time

    halting_generator = is_stable_generator(lower, time)

    return functools.partial(
        generator_front_end,
        halt_generator=halting_generator,)
```

Defines:
`make_threshold_time_halting_function`, never used.
Uses `activity` 45.

# 5 Iteration functions

## 5.1 Single iteration function

This function performs a single Diffusion Phase and a single Test Phase.

23 ⟨*synchronous iterate function* 23⟩≡ (80)

```
def synchronous_iterate(
    swarm,
    microtests,
    random_hypothesis_function,
    diffusion_function,
    test_phase_function=test_phase,
    multidiffusion=1,
    multitesting=1,
    multitest_function=all,
    rng=random,
):
    """\
Performs a synchronous iteration, one diffusion phase for all agents
followed by one test phase for all agents.

This function returns a generator which must be consumed once for each
iteration.\
"""

    while True:

        diffusion_phase_iterator = diffusion_function(
            swarm,
            random_hypothesis_function,
            multidiffusion,
            rng,)

        for _ in diffusion_phase_iterator:
            pass

        test_phase_iterator = test_phase_function(
            swarm=swarm,
            microtests=microtests,
            multitesting=multitesting,
            multitest_function=multitest_function,
            synchronous=True,
            rng=random,)

        for _ in test_phase_iterator:
            pass

        yield
```

Defines:
`synchronous_iterate`, used in chunks 30 and 46.
Uses `test_phase` 8.

## 5.2 Asynchronous iteration function

This function shuffles the swarm in place, that might not be acceptable eventually, in which case use `for agent in random.sample(swarm,len(swarm)`.

This function doesn't throw any exceptions, but I've not proved that it's running asynchronously.

I could make this system run more like the parallel implementation, where I call `update_state` rather than relying on the two synchronous phases to proceed in lock step.

25 ⟨*asynchronous iterate function* 25⟩≡ (80)

```
def asynchronous_iterate(
    swarm,
    microtests,
    random_hypothesis_function,
    diffusion_function,
    test_phase_function=test_phase,
    multidiffusion=1,
    multitesting=1,
    multitest_function=all,
    rng=random,
):
    """\
Performs an asynchronous iteration, all agents are selected in a
random order to perform one diffusion and one test in turn.

This function returns a generator which must be consumed once for each
iteration.\
"""

    while True:

        for agent in swarm:
            if agent.hypothesis is None:
                agent.hypothesis = random_hypothesis_function(random)

        random.shuffle(swarm)

        diffusion_phase_iterator = diffusion_function(
            swarm,
            random_hypothesis_function,
            multidiffusion,
            rng,)

        test_phase_iterator = test_phase_function(
            swarm=swarm,
            microtests=microtests,
            multitesting=multitesting,
            multitest_function=multitest_function,
```

```
            synchronous=False,
            rng=rng,)

        for _ in zip(diffusion_phase_iterator, test_phase_iterator):
            pass

        yield
```

Defines:
  `asynchronous_iterate`, never used.
Uses `test_phase` 8.

## 5.3 Parallel iteration function

I might want to check if there's a way of making this run like the asynchronous iteration, where I use the normal test phase and diffusion phase functions and just make them perform randomly.

27 ⟨*parallel iterate function* 27⟩≡ (80)

```
def parallel_iterate(
    swarm,
    microtests,
    random_hypothesis_function,
    diffusion_function,
    test_phase_function=test_phase,
    multidiffusion=1,
    multitesting=1,
    multitest_function=all,
    rng=random,
):
    """\
Performs a parallel iteration, all agents are updating their state (a
diffusion followed by a test) in parallel.

This function returns a generator which simply waits a short time
between yeilding, this can be used to ensure the parallel process
runs for a certain amount of wall clock time.\
"""

    context_free = diffusion_function is context_free_diffusion
    context_sensitive = diffusion_function is context_sensitive_diffusion

    scalar = test_phase_function is scalar_test_phase

    for agent in swarm:
        if agent.hypothesis is None:
            agent.hypothesis = random_hypothesis_function(random)

    worker_threads = (
        threading.Thread(
            target=update_state,
            args=(
                agent,
                swarm,
                random_hypothesis_function,
                context_free,
                context_sensitive,
                multidiffusion,
                microtests,
                multitesting,
                multitest_function,
                scalar,
```

```
                rng,
            )
        )
        for agent
        in swarm
    )

    for worker_thread in worker_threads:
        worker_thread.daemon = True
        worker_thread.start()

    while True:
        time.sleep(0.01)
        yield
```

Defines:
  parallel_iterate, used in chunk 29.

Uses context_free_diffusion 12, context_sensitive_diffusion 13, scalar_test_phase 9, test_phase 8, and update_state 29.

### 5.3.1 Update state function

29 ⟨*parallel update state function* 29⟩≡ (80)

```
def update_state(
    agent,
    swarm,
    random_hypothesis_function,
    context_free,
    context_sensitive,
    multidiffusion,
    microtests,
    multitesting,
    multitest_function,
    scalar,
    rng=random,
):
    """\
Repeatedly perform a diffusion and a test for an agent, with a short
sleep. This function is a helper function to parallel_iterate.\
"""

    while True:
        generic_single_agent_diffusion(
            agent,
            swarm,
            random_hypothesis_function,
            context_free,
            context_sensitive,
            multidiffusion,
            rng,
        )

        generic_single_agent_test(
            agent,
            swarm,
            microtests,
            scalar,
            multitesting,
            multitest_function,
        )

        sleepy_time = min(2,max(0,random.gauss(1,1)))
        time.sleep(sleepy_time)
```

Defines:
  `update_state`, used in chunk 27.
Uses `generic_single_agent_diffusion` 17, `generic_single_agent_test` 10, and `parallel_iterate` 27.

## 5.4 Run function

This function performs a given number of iterations and returns a list of all the clusters that have formed.

30 ⟨*run function* 30⟩≡ (80)

```
def run(
    swarm,
    microtests,
    random_hypothesis_function,
    max_iterations=1000,
    diffusion_function=passive_diffusion,
    halting_function=never_halt,
    halting_iterations=0,
    multitesting=1,
    multitest_function=all,
    report_iterations=10,
    test_phase_function=test_phase,
    hypothesis_string_function=str,
    max_cluster_report=None,
    iteration_function=synchronous_iterate,
    multidiffusion=1,
    rng=random,
):
    """\
The main front end to the SDS library. This will run forever until
manually halted, or until a halting condition is reached, afterwhich
it will return a collections.Counter of all the clusters.

:param swarm: A list of sds.Agent instances.
:param microtests: A collection of functions all of which take a hypothesis\
, as returned by random_hypothesis_function and return the result of a\
 microtest as either a scalar value or a boolean. All tests must \
return the same type.
:param random_hypothesis_function: A function which takes a random number \
generator and returns hypothesis suitable as input for all the \
functions in microtests.
:param max_iterations: The number of iterations afterwhich the SDS will \
halt.  If max_iterations=None, the SDS will never halt due to the num\
ber of iterations, but may be manually halted, or halted by the halti\
ng function.
:param diffusion_function: The diffusion function to use in the diffusion \
phase.
:param random: A random number generator, probably an instance of the rand\
om.Random class. Use an instance with an explicit seed to get repeata\
ble behaviour, else you may pass in the random module itself.
:param halting_function: (Default: never_halt) The function which takes \
the swarm as input and returns True if its condition is met.
:param halting_iterations: (Default: 0) The number of iterations between \
each call of the halting_function. If halting_iterations is a Falsy \
```

```
value (e.g. None, 0, False, [], ”) then the halting_function is \
never called.
:param multitesting: (Default: 1) The number of microtests each agent \
performs in the test phase. Must be an integer.
:param multitest_function: (Default: all) The function which takes a \
list of microtest results and turns them into a single result. The \
most likely values for multitest_function will be ’all’ i.e. all \
microtests must pass, and ’any’ i.e. at least one microtest must pass.
:param report_iterations: (Default: None) The number of iterations \
between each report to stdout of the hypotheses with the largest \
clusters.
:param test_phase_function: (Default: test_phase) The function to use in \
the test phase.
:param hypothesis_string_function: (Default: str) The function to call \
on a hypothesis to turn it into a string, suitable for inclusion in \
the report. If hypotheses are using built-in data types, str is \
often enough, otherwise a custom ’to string’ function must be supplied.
:param max_cluster_report: (Default: None) The maximum number of \
clusters to include in the report.
:param iteration_function: (Default: synchronous_iterate) The iteration \
function to call once per iteration.
:param multidiffusion: (Default: 1) The number of agents for a polling \
agent to poll during the diffusion phase. May be an integer or a \
float.\
"""

    try:

        iteration_generator = iteration_function(
            swarm=swarm,
            microtests=microtests,
            random_hypothesis_function=random_hypothesis_function,
            diffusion_function=diffusion_function,
            test_phase_function=test_phase_function,
            multidiffusion=multidiffusion,
            multitesting=multitesting,
            multitest_function=multitest_function,
            rng=rng,
        )

        for iteration_num, iteration in enumerate(iteration_generator):

            if (
                report_iterations
                and iteration_num % report_iterations == 0
            ):

                clusters = count_clusters(swarm)
```

```
                agent_count = len(swarm)

                print("{i:4} Activity: {a:0.3f}. {c}".format(
                    i=iteration_num,
                    a=sum(clusters.values())/float(agent_count),
                    c=", ".join(
                        "{hyp}:{count}".format(
                            hyp=hypothesis_string_function(hyp),
                            count=count
                        )
                        for hyp,count
                        in clusters.most_common(max_cluster_report)
                    ),
                ))

            if(
                (
                    halting_iterations
                    and iteration_num % halting_iterations == 0
                    and halting_function(swarm)
                ) or (
                    max_iterations and iteration_num >= max_iterations
                )
            ):

                break

    except KeyboardInterrupt:

        pass

    return count_clusters(swarm)
```

Defines:
`run`, used in chunks 34, 36b, 37, 40–43, 47, 61, 64, 67, 73, and 75.
Uses `Agent` 4, `count_clusters` 44a, `never_halt` 19b, `passive_diffusion` 11, `synchronous_iterate` 23, and `test_phase` 8.

32 ⟨*library dependencies* 5c⟩+≡ (80) ◁5c 33b▷

```
import itertools
```

## 5.5 Write swarm to file

33a ⟨*write swarm function* 33a⟩≡ (80)

```
def write_swarm(swarm, outfile):
    """\
Writes a swarm to a file-like object.\
"""
    json.dump(
        {
            'agent count':len(swarm),
            'clusters':count_clusters(swarm).most_common(),
        },
        outfile,
    )
```

Defines:
  `write_swarm`, used in chunk 34.
Uses `count_clusters` 44a.

33b ⟨*library dependencies* 5c⟩+≡ (80) ◁32 36a▷

```
import json
```

## 5.6 Continuous loop function

34 ⟨*run daemon* 34⟩≡ (80)

```
def run_daemon(
    swarm,
    microtests,
    random_hypothesis_function,
    diffusion_function,
    max_iterations=None,
    halting_function=never_halt,
    halting_iterations=0,
    multitesting=1,
    multitest_function=all,
    report_iterations=None,
    test_phase_function=test_phase,
    hypothesis_string_function=str,
    max_cluster_report=None,
    out_file_name='/tmp/clusters.json',
    rng=random,
):
    """\
Calls sds.run in a daemon thread. The daemon can be interacted with
through the command line.

q: Halt the SDS and kill the daemon.
c: Print the largest clusters to the screen.
w: Write the largest clusters to file.

Anything else is printed to stdout.\
"""

    def write_status(swarm):
        with open(out_file_name,'w') as f:
            write_swarm(swarm,f)
            print('wrote swarm status to',out_file_name)

    control_queue = queue.Queue()
    control_queue.put(True)

    def swarm_iterator():

        print('starting SDS')
        run(
            swarm,
            microtests,
            random_hypothesis_function,
            max_iterations,
            diffusion_function,
            halting_function,
            halting_iterations,
```

```
            multitesting,
            multitest_function,
            report_iterations,
            test_phase_function,
            hypothesis_string_function,
            max_cluster_report,
            rng,)

        print('finishing SDS')

        write_status(swarm)

        control_queue.task_done()

    t = threading.Thread(target=swarm_iterator)
    t.daemon = True # Program will exit when only daemons are left.
    t.start()
    del t

    def interface_manager():

        while True:

            instr = input()

            if instr == 'q':
                print("'q' received quitting")
                control_queue.task_done()
                break
            elif instr == 'c':
                print(count_clusters(swarm).most_common(max_cluster_report))
            elif instr == 'w':
                write_status(swarm)
            else:
                print('You said:',instr.upper())

    t = threading.Thread(target=interface_manager)
    t.daemon = True # Program will exit when only daemons are left.
    t.start()
    del t

    control_queue.join()

    print('done run_daemon')

    return count_clusters(swarm)
```

Defines:
  `run_daemon`, used in chunk 62.
Uses `count_clusters` 44a, `never_halt` 19b, `run` 30, `test_phase` 8, and `write_swarm` 33a.

36a ⟨*library dependencies* 5c⟩+≡ (80) ◁33b 44b▷

```
import queue
import time # for time.sleep
import itertools # for itertools.count
import threading # for threading.Thread
```

# 6 Coupled SDS

## 6.1 Coupled diffusion

36b ⟨*coupled diffusion* 36b⟩≡ (80)

```
def coupled_diffusion(
    swarms,
    random,
    random_hypothesis_functions,
    diffusion_functions,
):
    """\
Performs Coupled diffusion when passed a list of swarms, a list of
random hypothesis functions and a list of diffusion functions. Not
tested with the newest version of sds.run.\
"""
    for swarm, diffusion_function, random_hypothesis_function in zip(
        swarms,
        diffusion_functions,
        random_hypothesis_functions
    ):
        diffusion_function(swarm, random, random_hypothesis_function)
```

Defines:
`coupled_diffusion`, used in chunk 42.
Uses `run` 30.

## 6.2 Coupled test phase

Multitesting is weird for a coupled test phase, as each test will involve more random agents, so after ten tests you've hit at least ten agents, and it would be strange to set them all inactive if a couple were good, or active if lots were bad.

This is of course not weird if you have a component hypothesis, and a data swarm, and a number of partial evaluations, then multitesting would be to select a hypothesis, and a data point, and to test it against multiple microtests. It's possible that it only seems strange because in hyperplane parameter estimation there is only one test. An alternative is to perform the test with the hypothesis and different data points, and to update the data point with the result of each test, but only update the hypothesis with the result of the combined tests.

Scalar testing can be done when multitesting equals 1, but I'll have to record the randomly selected agents for each master agent.

Now, is it the case that you don't need multitesting if you have coupled sds? Multitesting effectively increases or decreases the test score, but coupled sds weights the test selection.

Maybe there's a decision to be made, if I had multitesting equal to 10, and that meant two hypothesis agents and two data agents, I'd affect the activity of 31 agents. So which of these options actually makes sense? The activity of the test which made the change is impossible as in the "all" case or "any" case I can't tell which one is responsible.

I think the activity of the master agent should be pushed onto all of the other agents queried in the multitesting. But I don't think I can be arsed to implement it now.

37 ⟨*coupled test phase* 37⟩≡ (80)

```
def generic_coupled_test_phase(
    master_swarm_num,
    swarms,
    random,
    multitesting,
    multitest_function,
    microtests,
    scalar,
):
    """\
Performs Coupled test when passed the index of a master swarm, a list
of swarms, and a list of lists of microtests. Not tested with the
latest version of sds.run.\
"""

    if not multitesting == 1:
        raise NotImplementedError(
            "Sorry, I've not got around to multitesting for coupled "
            "sds yet.")

    test_results = []
```

```
    tested_agents = []

    for master_agent in swarms[master_swarm_num]:

        agents = [random.choice(swarm) for swarm in swarms]

        agents[master_swarm_num] = master_agent

        hypotheses = tuple(agent.hypothesis for agent in agents)

        microtest = random.choice(microtests)

        test_results.append(microtest(*hypotheses))

        tested_agents.append(agents)

    if False and scalar:

        test_results = [
            test_result > random.choice(test_results)
            for test_result
            in test_results
        ]

    for test_result, agents in zip(test_results, tested_agents):

        for agent in agents:

            agent.active = test_result
```

Defines:
`generic_coupled_test_phase`, used in chunks 40 and 41.
Uses `run` 30.

This is the abandoned code chunk for coupled sds multitesting.

39 ⟨*coupled multitesting* 39⟩≡

```
test_results = []

for test_num in range(multitesting):

    agents = [random.choice(swarm) for swarm in swarms]

    agents[master_swarm_num] = master_agent

    hypotheses = tuple(agent.hypothesis for agent in agents)

    microtest = random.choice(microtests)

    test_results.append(microtest(*hypotheses))

result = multitest_function(test_results)

for agent in agents:

    agent.active = result
```

### 6.2.1 Synchronous coupled test phase

40 ⟨*synchronous coupled test phase* 40⟩≡ (80)

```
# master/slave synchronisation
def synchronous_coupled_test_phase(
    swarms,
    random,
    multitesting,
    multitest_function,
    microtests,
    scalar,
):
    """\
Perform a Synchronous coupled test phase. Not tested with the latest
version of sds.run.\
"""

    master_swarm_num = 0

    generic_coupled_test_phase(
        master_swarm_num,
        swarms,
        random,
        multitesting,
        multitest_function,
        microtests,
        scalar,
    )
```

Defines:
  `synchronous_coupled_test_phase`, used in chunk 65.
Uses `generic_coupled_test_phase` 37 and `run` 30.

### 6.2.2 Sequential coupled test phase

41 ⟨*sequential coupled test phase* 41⟩≡ (80)

```
def sequential_coupled_test_phase(
    swarms,
    random,
    multitesting,
    multitest_function,
    microtests,
    scalar,
):
    """\
Perform a Sequential master coupled test phase. Not tested with the latest
version of sds.run.\
"""

    for master_swarm_num in range(len(swarms)):

        generic_coupled_test_phase(
            master_swarm_num,
            swarms,
            random,
            multitesting,
            multitest_function,
            microtests,
            scalar,
        )
```

Defines:
`sequential_coupled_test_phase`, used in chunk 65.
Uses `generic_coupled_test_phase` 37 and `run` 30.

## 6.3 Coupled iteration

42 ⟨*coupled iterate* 42⟩≡ (80)

```
def iterate_coupled(
    swarms,
    random_hypothesis_functions,
    diffusion_functions,
    random,
    multitesting,
    multitest_function,
    report_iterations,
    test_phase_function,
    microtests,
    scalar,
):
    """\
Perform an iteration of Coupled SDS. Not tested with the latest
version of sds.run.\
"""

    coupled_diffusion(
        swarms,
        random,
        random_hypothesis_functions,
        diffusion_functions)

    test_phase_function(
        swarms,
        random,
        multitesting,
        multitest_function,
        microtests,
        scalar,
    )
```

Defines:
`iterate_coupled`, used in chunk 43.
Uses `coupled_diffusion` 36b and `run` 30.

43 ⟨*coupled run* 43⟩≡ (80)

```
def run_coupled(
    swarms,
    random_hypothesis_functions,
    max_iterations,
    diffusion_functions,
    random,
    multitesting,
    multitest_function,
    report_iterations,
    test_phase_function,
    hypothesis_string_function,
    max_cluster_report,
    microtests,
    scalar,
):
    """\
Perform a Coupled SDS. Not tested with the latest version of sds.run.\
"""

    if max_iterations is None:

        iterator = itertools.count()

    else:

        iterator = range(max_iterations)

    try:

        for iteration in iterator:

            iterate_coupled(
                swarms,
                random_hypothesis_functions,
                diffusion_functions,
                random,
                multitesting,
                multitest_function,
                report_iterations,
                test_phase_function,
                microtests,
                scalar,
            )

            if report_iterations and iteration % report_iterations == 0:

                clusters_list = tuple(count_clusters(swarm) for swarm in swarms)
```

```
                agent_counts = tuple(len(swarm) for swarm in swarms)

                active_count = tuple(
                    sum(clusters.values())
                    for clusters
                    in clusters_list)

                print(agent_counts,active_count,clusters_list)

    except KeyboardInterrupt:

        pass

    return tuple(count_clusters(swarm) for swarm in swarms)
```

Defines:
`run_coupled`, used in chunk 65.
Uses `count_clusters` 44a, `iterate_coupled` 42, and `run` 30.

# 7 Analysis functions

## 7.1 Count clusters function

This function returns a list of all the clusters in the swarm.

44a ⟨*count clusters function* 44a⟩≡ (80)

```
def count_clusters(swarm):
    """\
Returns the number of active agents at each hypothesis with at least one
active agent as a collections.Counter.\
"""

    return collections.Counter(
        agent.hypothesis
        for agent
        in swarm
        if agent.active
    )
```

Defines:
`count_clusters`, used in chunks 30, 33a, 34, 43, and 59.

44b ⟨*library dependencies* 5c⟩+≡ (80) ◁36a 56b▷

```
import collections
```

## 7.2 Calculate Activity

45 ⟨*activity function* 45⟩≡ (80)

```
def activity(swarm):
    """\
Return the proportion of the swarm which are active between 0 and 1.\
"""

    agent_count = len(swarm)

    active_count = sum(1 for agent in swarm if agent.active)

    return active_count/agent_count
```

Defines:
`activity`, used in chunks 10, 20–22, and 46.

### 7.3 Estimate Uniform Background Noise

46 ⟨*estimate noise* 46⟩≡ (80)

```
def estimate_noise(
    microtests,
    random_hypothesis_function,
    noise_agent_count=100,
    iterations=100,
):
    """\
Returns an estimate of the uniform background noise, between 0 and 1.\
"""

    def no_diffusion(swarm, random, random_hypothesis_function):
        for agent in swarm:
            agent.active = False
            agent.hypothesis = random_hypothesis_function(random)

    noise_swarm = Agent.initialise(100)

    activities = []

    for iteration in range(iterations):
        synchronous_iterate(
            noise_swarm,
            microtests,
            random_hypothesis_function,
            no_diffusion,
            random,)

        activities.append(activity(noise_swarm))

    return sum(activities)/iterations
```

Defines:
`estimate_noise`, never used.
Uses `activity` 45, `Agent` 4, `initialise` 5a, and `synchronous_iterate` 23.

## 7.4 Swarm from clusters

A full swarm can be recovered from a `clusters` object and a count of the total number of agents.

47 ⟨*swarm from clusters* 47⟩≡ (80)

```
def swarm_from_clusters(agent_count, clusters):
    """\
Returns a swarm suitable for use in functions like sds.run.

Clusters should be a dictionary or collections.Counter of the
hypotheses of active agents.\
"""

    active_agents = (
        (
            Agent(hypothesis=hyp,active=True)
            for _ in
            range(count)
        )
        for hyp, count
        in clusters.items())

    inactive_count = agent_count - sum(clusters.values())

    return (
        Agent.initialise(inactive_count)
        + list(itertools.chain.from_iterable(active_agents)))
```

Defines:
`swarm_from_clusters`, never used.
Uses `Agent` 4, `initialise` 5a, and `run` 30.

## 7.5 Pretty printing

This function renders a list of clusters into text.

48 ⟨*pretty print clusters with values* 48⟩≡ (80)

```
def pretty_print_with_values(clusters, search_space, max_clusters=None):

    string_template = "{c:6d} at hyp {h:6d} (value: {e:0.6f})"

    cluster_strings = [
        string_template.format(
            c=count,
            h=hyp,
            e=search_space[hyp])
        for hyp, count
        in clusters.most_common(max_clusters)
    ]

    return "\n".join(cluster_strings)
```

Defines:
`pretty_print_with_values`, never used.

# 8 Multilayer SDS

## 8.1 Swarm class

Multilayer SDS mostly uses functions which are very similar to Standard SDS, but a `Swarm` class is also required to associate each swarm of agents with a particular function for generating a random hypothesis.

49 ⟨*swarm class* 49⟩≡ (80)

```
class Swarm:
    """
A multilayer SDS swarm. Best instansiated with make_ml_sds."""

    ⟨swarm initialisation function 50a⟩

    ⟨swarm passive diffusion function 50b⟩

    ⟨swarm context free diffusion function 51⟩

    ⟨swarm context sensitive diffusion function 52⟩

    ⟨swarm test function 53a⟩

    ⟨swarm iterate function 53b⟩

    ⟨swarm set activity function 54b⟩

    ⟨swarm set hypothesis function 54a⟩
```

Uses `make_ml_sds` 55.

**Swarm initialisation function**

50a ⟨*swarm initialisation function* 50a⟩≡ (49)

```
def __init__(self, size, random_hypothesis_function, lower_layer=None):

    self.agents = [
        Agent(active=False, hypothesis=None)
        for _
        in range(size)
    ]

    if lower_layer is None:

        lower_layer = []

    self.lower_layer = lower_layer

    self.random_hypothesis = random_hypothesis_function
```

Uses `Agent` 4.

**Swarm diffusion function** This implements passive diffusion for Multilayer SDS. It's not clear whether this could be modified to be identical to the functions written for SDS, or if the other Diffusion Phase variants need to be reimplemented for Multilayer SDS.

50b ⟨*swarm passive diffusion function* 50b⟩≡ (49)

```
@staticmethod
def passive_diffusion(swarm, solitarity, random):

    for agent_num, agent in enumerate(swarm.agents):

        if not agent.active:

            polled_agent = random.choice(swarm.agents)

            if polled_agent.active and random.random() > solitarity:

                swarm.set_hypothesis(agent_num, polled_agent.hypothesis)

            else:

                agent.hypothesis = swarm.random_hypothesis(
                    agent_num,
                    random,
                )
```

Uses `passive_diffusion` 11.

51 ⟨*swarm context free diffusion function* 51⟩≡ (49)

```
@staticmethod
def context_free_diffusion(swarm, random):

    for agent_num, agent in enumerate(swarm.agents):

        polled_agent = random.choice(swarm.agents)

        if agent.active:

            if polled_agent.active:

                swarm.set_activity(agent_num, False)

                agent.hypothesis = swarm.random_hypothesis(
                    agent_num,
                    random,
                )

        else:

            if polled_agent.active:

                swarm.set_hypothesis(agent_num, polled_agent.hypothesis)

            else:

                agent.hypothesis = swarm.random_hypothesis(
                    agent_num,
                    random,
                )
```

Uses `context_free_diffusion` 12.

52 ⟨*swarm context sensitive diffusion function* 52⟩≡ (49)

```
@staticmethod
def context_sensitive_diffusion(swarm, random):

    for agent_num, agent in enumerate(swarm.agents):

        polled_agent = random.choice(swarm.agents)

        if agent.active:

            if (
                polled_agent.active
                and (agent.hypothesis == polled_agent.hypothesis)
            ):


                swarm.set_activity(agent_num, False)

                agent.hypothesis = swarm.random_hypothesis(
                    agent_num,
                    random,
                )

        else:

            if polled_agent.active:

                swarm.set_hypothesis(agent_num, polled_agent.hypothesis)

            else:

                agent.hypothesis = swarm.random_hypothesis(
                    agent_num,
                    random,
                )
```

Uses `context_sensitive_diffusion` 13.

**Swarm test function** The test function is similar to the standard Test Phase only the result of the test is propagated recursively to all connected agents.

53a ⟨*swarm test function* 53a⟩≡ (49)

```
def test(self, microtests, random, multitest=1, multitest_fun=None):

    for num, agent in enumerate(self.agents):

        microtest = random.choice(microtests)

        if multitest == 1:

            self.set_activity(num, microtest(agent.hypothesis))

        else:

            self.set_activity(
                num,
                multitest_fun(
                    random.choice(microtests)(agent.hypothesis)
                    for _
                    in range(multitest)
                )
            )
```

**Swarm iterate function** This is a convenience function which calls one diffusion phase followed by one test phase.

53b ⟨*swarm iterate function* 53b⟩≡ (49)

```
def iterate(
    self,
    microtests,
    random,
    diffusion_function=passive_diffusion.__func__,
    multitest=1,
    multitest_fun=None,
    solitarity=1,
):
    diffusion_function(self, solitarity, random)
    self.test(microtests, random, multitest, multitest_fun)
```

Uses `passive_diffusion` 11.

### Swarm set hypothesis function

54a ⟨*swarm set hypothesis function* 54a⟩≡ (49)

```
def set_hypothesis(self, agent_num, new_hypothesis):

    self.agents[agent_num].hypothesis = new_hypothesis

    if len(self.lower_layer) == 0:
        return

    for lower_swarm, hypothesis_component in (
        zip(self.lower_layer,new_hypothesis)):

        lower_swarm.set_hypothesis(agent_num, hypothesis_component)
```

### Swarm set activity function

54b ⟨*swarm set activity function* 54b⟩≡ (49)

```
def set_activity(self, agent_num, new_activity):

    self.agents[agent_num].active = new_activity

    for swarm in self.lower_layer:

        swarm.set_activity(agent_num, new_activity)
```

## 8.2 Multilayer SDS factory function

To properly implement Multilayer SDS the random hypothesis generation functions recursively call the random hypothesis generation functions of lower layers, and so a convenience function has been developed that takes an intended topology of swarms and composes all the required random hypothesis generation functions.

55 ⟨*make multilayer sds function* 55⟩≡ (80)

```
def make_ml_sds(swarm_size, bottom_hyp_functions, topology):

    lower_layer = [
        Swarm(
            size=swarm_size,
            random_hypothesis_function=hyp_fun
        )
        for hyp_fun
        in bottom_hyp_functions
    ]

    for layer_num, swarm_splits in enumerate(topology,start=1):

        ⟨construct new layer 56a⟩

        lower_layer = layer

    top_swarm = lower_layer[0]

    return top_swarm
```

Defines:
`make_ml_sds`, used in chunks 49 and 59.

56a ⟨*construct new layer* 56a⟩≡ (55)

```
layer = []

swarm_offset = 0

for split_num, swarm_split in enumerate(swarm_splits):

    lower_layer_start = swarm_offset

    lower_layer_end = swarm_offset+swarm_split

    random_hypothesis_function = functools.partial(
        random_compound_hyp,
        lower_layer[lower_layer_start:lower_layer_end],
    )

    new_swarm = Swarm(
        size=swarm_size,
        lower_layer=lower_layer[lower_layer_start:lower_layer_end],
        random_hypothesis_function=random_hypothesis_function)

    layer.append(new_swarm)

    swarm_offset += swarm_split
```

56b ⟨*library dependencies* 5c⟩+≡ (80) ◁44b 58b▷

```
import functools
```

## 8.3 Random hypothesis, and single diffusion functions

Multilayer SDS requires that a single agent can perform the diffusion process, not the whole swarm, this is the purpose of the `single_diffusion` function. It also requires that agents in higher level swarms can randomly generate a hypothesis by calling the diffusion function on each of the associated agents in the swarms in the lower layer, this is the purpose of the `random_compound_hyp` function.

### Single diffusion function

57a ⟨*single diffusion function* 57a⟩≡ (80)

```
def single_diffusion(agent_num, swarm, random):

    # agent is guaranteed to be inactive
    agent = swarm.agents[agent_num]

    polled_agent = random.choice(swarm.agents)

    if polled_agent.active:

        swarm.set_hypothesis(agent_num,polled_agent.hypothesis)

        return polled_agent.hypothesis

    else:

        new_hyp = swarm.random_hypothesis(agent_num, random)

        agent.hypothesis = new_hyp

        return new_hyp
```

### Random compound hypothesis function

57b ⟨*random compound hypothesis function* 57b⟩≡ (80)

```
def random_compound_hyp(lower_swarms, num, random):

    return tuple(
        single_diffusion(num, lower_swarm, random)
        for lower_swarm
        in lower_swarms
    )
```

### 8.4 Flatten hypothesis function

Hypotheses generated by Multilayer SDS can be very deeply nested lists, this function allows the removal of some of the levels of nesting, until a single flat list is returned. A flat list will be returned if the value if `times=len(topology)-1`.

58a ⟨*flatten hypothesis* 58a⟩≡ (80)

```
def flatten_hypothesis(hypothesis,times):
    new_hypothesis = itertools.chain.from_iterable(hypothesis)
    if times == 1:
        return list(new_hypothesis)
    else:
        return flatten_hypothesis(new_hypothesis,times-1)
```

58b ⟨*library dependencies* 5c⟩+≡ (80) ◁56b 68▷

```
import itertools
```

### 8.5 Concurrent Execution

Python has some libraries for concurrent execution, this investigation is to determine if SDS can be execued concurrently to improve performance.

This is not an investigation into the absolute performance itself, as Python is not the optimal language for speed of execution, simply to discover which architectures, if any, offer significant performance improvements.

Python has different methods for when the task is limited by speed of execution (CPU bound), or speed of data transfer (I/O bound). If it's CPU bound you need to use multi-processing, multithreading will work fine for I/O bound tasks.

"The strength of threads is shared state. The weakness of threads is shared state (managing race conditions)." From `https://dl.dropboxusercontent.com/u/3967849/pyru/_build/html/int` Thinking about Concurrency, Raymond Hettinger, Python core developer.

Some tasks will be soluable within a single thread. More complex problems will need multiple cores, truly large problems need distributed processing. Second, "multiple cores" category is becoming less relevant in this world. Don't spend too much time on it, if the real issue is in much larger problems.

When you have threads that end, you join the threads, when you have deamon threads which never terminate, you join the message queue.

Queues are only suitable for when your underlying data flow is a directed acyclic graph.

A good development strategy is to first use "map" to test the code in a single process and a single thread before moving to parallel execution.

## 8.6 MSDS Example

59 ⟨*multilayer sds example* 59⟩≡

```
import sds # Import sds library
import random
r = random.Random() # Initialise random number generator

# Initialise a list of one function per swarm, they take agent_num, and
# random, and return a hypothesis component.
random_hyp_functions = [
    lambda num, random: random.randrange(10),
    lambda num, random: random.randrange(10),
    lambda num, random: random.randrange(10),
    lambda num, random: random.randrange(10),
]

# Declare a topology, starting with a list of one [[1]] for each swarm,
# make a list of lists of ints, where each list of ints means how the
# previous numbers will be divided.
topology = [[2,2],[2]]

# Create a top swarm, with all its related infrastructure.
swarm = sds.make_ml_sds(
    swarm_size=100,
    bottom_hyp_functions=random_hyp_functions,
    topology=topology,
)

⟨initialise search space 60a⟩
⟨initialise microtests 60b⟩

# Iterate the swarm 100 times.
for _ in range(100):
    swarm.iterate(
        microtests=microtests,
        random=r,
    )

# You can use the free count_clusters function on Swarm.agents.
clusters = sds.count_clusters(swarm.agents)

print(clusters.most_common(10))
```

Uses `count_clusters` 44a and `make_ml_sds` 55.

The search space is a 4 dimensional array of random numbers in the interval $[-1, 1]$.

60a ⟨*initialise search space* 60a⟩≡ (59)

```
search_space = [
    [
        [
            [
                r.uniform(-1,1)
                for _
                in range(10)
            ]
            for _
            in range(10)
        ]
        for _
        in range(10)
    ]
    for _
    in range(10)
]
```

A list of microtests, they take a hyp, and return a boolean. This isn't a member variable of Swarm as only the top swarm is involved in testing.

60b ⟨*initialise microtests* 60b⟩≡ (59)

```
microtests = [
    lambda h: True,
    lambda h: False,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.1,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.2,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.3,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.4,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.5,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.6,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.7,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.8,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 0.9,
    lambda h: search_space[h[0][0]][h[0][1]][h[1][0]][h[1][1]] > 1,
]
```

# 9 String search implemented with the library

61 ⟨*src/string-search.py* 61⟩≡ (85c)

```
import random
import functools
import sds

search_space = "xxhellxelloxhexhelxoxxxhxlloxxx"
model = "hello"

def random_hyp(rnd): return rnd.randint(0,len(search_space)-len(model))

def make_microtest(offset):
    return lambda hyp: search_space[hyp+offset] == model[offset]

microtests = [make_microtest(offset) for offset in range(len(model))]

clusters = sds.run(
    swarm=sds.Agent.initialise(agent_count=1000),
    microtests=microtests,
    random_hypothesis_function=random_hyp,
    max_iterations=300,
    diffusion_function=sds.passive_diffusion,
    random=random.Random(),
    report_iterations=10,
)

print(clusters.most_common())
```

Uses `Agent` 4, `initialise` 5a, `passive_diffusion` 11, and `run` 30.

## 10 SDS Daemon implemented with the library

62 ⟨*src/sds-daemon.py* 62⟩≡

```
import functools
import random
import sds

search_space, model = "xxhexlxelloxhexhxlxoxxxhxllxxxx","hello"

def random_hyp(rnd):
    return rnd.randint(0,len(search_space)-len(model))

def test_hyp(offset, hyp):
    return search_space[hyp+offset] == model[offset]

microtests = [
    functools.partial( test_hyp, offset)
    for offset
    in range(len(model))
]

swarm = sds.Agent.initialise(agent_count=1000)

max_iterations = None

diffusion_function=sds.passive_diffusion

sds.run_daemon(
    swarm,
    microtests,
    random_hyp,
    diffusion_function,
    max_iterations,
    halting_function=sds.never_halt,
    halting_iterations=0,
    multitesting=1,
    multitest_function=all,
    report_iterations=100,
    test_phase_function=sds.test_phase,
    hypothesis_string_function=str,
    max_cluster_report=None,
    out_file_name='./daemon-clusters.json',
    random=random,
)

print('done src/sds-daemon.py')
```

Uses `Agent` 4, `initialise` 5a, `never_halt` 19b, `passive_diffusion` 11, `run_daemon` 34, and `test_phase` 8.

# 11 Data-driven SDS implemented with the library

This implements data driven sds. It has the problem that you can reach strong stability with imperfect hypotheses.

See this final state, where `model = 'hello'` and `hypothesis 6 == 'xello'`.

```
[(CompHyp(hyp=6, data=3), 381), (CompHyp(hyp=6, data=2), 296),
 (CompHyp(hyp=6, data=4), 171), (CompHyp(hyp=6, data=1), 152),]
[(6, 1000)]
```

This shows that all of the 1000 agents have converged on hypothesis 6. They are also maintaining the data points `[1,2,3,4]`, i.e. not 0, which is the only test which fails.

64 ⟨*src/data-driven.py* 64⟩≡

```
import random, functools, sds
from collections import namedtuple, Counter

search_space, model = "xxhexlxelloxhexhxlxoxxxhxllxxxx","hello"

CompHyp = namedtuple('CompHyp',('hyp','data'))

def make_microtest(offset):
    return lambda hyp: search_space[hyp+offset] == model[offset]

microtests = [make_microtest(offset) for offset in range(len(model))]

def random_hyp(rnd): return rnd.randint(0,len(search_space)-len(model))

def data_driven_random_hyp(rnd):
    return CompHyp(
        hyp=random_hyp(rnd),
        data=rnd.randrange(len(microtests)),)

def data_driven_microtest(swarm, compound_hyp):
    random_agent = random.choice(swarm)

    result = microtests[random_agent.hypothesis.data](compound_hyp.hyp)

    random_agent.active = result

    return result

swarm = sds.Agent.initialise(agent_count=1000)

clusters = sds.run(
    swarm,
    microtests=[functools.partial(data_driven_microtest,swarm)],
    random_hypothesis_function=data_driven_random_hyp,
    max_iterations=300,
    diffusion_function=sds.passive_diffusion,
    random=random,
)

no_data_clusters = Counter()
[no_data_clusters.update({hyp:count}) for (hyp,data), count in clusters.items()]
print(clusters.most_common(), no_data_clusters.most_common())
```

Uses `Agent` 4, `initialise` 5a, `passive_diffusion` 11, and `run` 30.

## 12 Coupled SDS implemented with the library

65 ⟨*src/coupled-sds.py* 65⟩≡

```
import itertools, random, sds

swarms = [
    sds.Agent.initialise(agent_count=100),
    sds.Agent.initialise(agent_count=101),]

search_space, model = "xxhexlxelloxhexhxlxoxxxhxllxxxx","hello"

microtests = [ lambda loc,data: search_space[loc+data] == model[data] ]

random_hypothesis_functions = [
    lambda rnd: rnd.randint(0,len(search_space)-len(model)),
    lambda rnd: rnd.randrange(len(model)),
]

diffusion_functions = (
    sds.passive_diffusion,
    sds.context_sensitive_diffusion,)

test_phase_functions = (
    sds.synchronous_coupled_test_phase,
    sds.sequential_coupled_test_phase,)

is_scalars = (False, True)

for test_phase_function, scalar in itertools.product(
    test_phase_functions,
    is_scalars,
):

    clusters = sds.run_coupled(
        swarms=swarms,
        microtests=microtests,
        random_hypothesis_functions=random_hypothesis_functions,
        max_iterations=100,
        diffusion_functions=diffusion_functions,
        random=random,
        multitesting=1,
        multitest_function=all,
        report_iterations=None,
        test_phase_function=test_phase_function,
        hypothesis_string_function=None,
        max_cluster_report=None,
        scalar=scalar,
    )
```

```
    print(clusters)
```

Uses `Agent` 4, `context_sensitive_diffusion` 13, `initialise` 5a, `passive_diffusion` 11, `run_coupled` 43, `sequential_coupled_test_phase` 41, and `synchronous_coupled_test_phase` 40.

## 13 SDS Simulator implemented with the library

67 ⟨*sds simulator 67*⟩≡ (80)

```
def simulate(
    scores,
    max_iterations=1000,
    report_iterations=500,
    diffusion_function=passive_diffusion,
    agent_count=1000,
    multitesting=1,
    multitest_function=all,
    random=random,
    random_hyp=None,
    halting_function=never_halt,
    halting_iterations=None,
):

    if random_hyp is None:

        def random_hyp(rnd): return rnd.randrange(1,len(scores))

    if halting_iterations is None:
        halting_iterations = report_iterations

    def make_microtest(test_num, rnd):
        return lambda hyp: rnd.random() < scores[test_num]

    microtests = [
        lambda hyp: random.random() < scores[hyp]
    ]

    swarm=Agent.initialise(agent_count=agent_count)

    swarm[0].active = True
    swarm[0].hypothesis = 0

    clusters = run(
        swarm=swarm,
        microtests=microtests,
        random_hypothesis_function=random_hyp,
        max_iterations=max_iterations,
        diffusion_function=passive_diffusion,
        multitesting=multitesting,
        multitest_function=multitest_function,
        random=random,
        report_iterations=report_iterations,
        halting_function=halting_function,
        halting_iterations=halting_iterations,
    )
```

```
    return clusters
```

Defines:
`simulate`, used in chunk 69.
Uses `Agent` 4, `initialise` 5a, `never_halt` 19b, `passive_diffusion` 11, and `run` 30.

68 ⟨*library dependencies* 5c⟩+≡ (80) ◁58b

```
import random
```

## 13.1 SDS Simulator front end

69 ⟨*src/simulator.py* 69⟩≡

```
import random
import functools
import sds
import argparse

if __name__ == '__main__':

    parser = argparse.ArgumentParser(
        description="Simulates SDS",
        epilog="",
    )

    parser.add_argument(
        "-s","-scores",
        nargs='+',
        required=True,
        type=float,
        help="Scores of the hypotheses",
    )

    parser.add_argument(
        "-e","-seed",
        type=int,
        help="Random seed",
    )

    parser.add_argument(
        "-i","-iterations",
        type=int,
        default=1000,
        help="Maximum iterations",
    )

    parser.add_argument(
        "-r","-report-iterations",
        type=int,
        default=None,
        help="Number of iterations between every analysis",
    )

    parser.add_argument(
        "-n","-agent-count",
        type=int,
        default=1000,
        help="Number of agents in the swarm",
    )
```

```
diffusion_types = [
    'passive',
    'context-free',
    'context-sensitive',
]
diffusion_functions = [
    sds.passive_diffusion,
    sds.context_free_diffusion,
    sds.context_sensitive_diffusion,
]

parser.add_argument(
    "-d","-diffusion",
    choices=diffusion_types,
    default=diffusion_types[0],
    help="Type of diffusion phase",
)

parser.add_argument(
    "-p","-prob",
    default=0,
    type=str,
    help="Probability of picking optimal hypothesis",
)

halting_types = [
    'never',
    'weak',
    'threshold',
]
halting_functions = [
    sds.never_halt,
    sds.make_stability_halting_function,
    sds.make_instant_threshold_halt_function,
]
parser.add_argument(
    '-a','-halt',
    choices=halting_types,
    default=halting_types[0],
    help='Type of halting',
)
parser.add_argument(
    '-l','-lower',
    type=float,
    default=0.5,
    dest='halting_lower_bound',
    help="Activity lower bound for stability halting",
)
```

```
parser.add_argument(
    '-g','-region',
    type=float,
    default=0.25,
    dest='halting_stability_region',
    help="Activity stability region for stability halting",
)
parser.add_argument(
    '-t','-time',
    type=int,
    default=32,
    dest='halting_stability_time',
    help="Activity stability time for stability halting",
)

args = parser.parse_args()

print(args)

def random_hyp_fun(P, score_len, rnd):
    if rnd.random() < P:
        return 0
    else:
        return rnd.randrange(1,score_len)

if args.prob == 'even':
    args.random_hyp = lambda rnd: rnd.randrange(len(args.scores))
else:
    args.random_hyp = lambda rnd: random_hyp_fun(float(args.prob), len(args.scores),


if args.report_iterations is None:
    args.report_iterations = args.iterations // 10

args.diffusion = dict(
    zip(diffusion_types,diffusion_functions)
)[args.diffusion]

halting_dict = dict(zip(halting_types,halting_functions))
halting_function = halting_dict[args.halt]
if args.halt == 'never':
    pass
elif args.halt == 'weak':
    halting_function = halting_function(
        lower=args.halting_lower_bound,
        region=args.halting_stability_region,
        time=args.halting_stability_time,)
elif args.halt == 'threshold':
    halting_function = halting_function(
```

```
        threshold=args.halting_lower_bound,)

    print(args)

    clusters = sds.simulate(
        args.scores,
        random=random.Random(args.seed),
        max_iterations=args.iterations,
        report_iterations=args.report_iterations,
        diffusion_function=args.diffusion,
        agent_count=args.agent_count,
        random_hyp=args.random_hyp,
        halting_function=halting_function,
    )

    print(clusters.most_common())
```

Uses `context_free_diffusion` 12, `context_sensitive_diffusion` 13, `make_instant_threshold_halt_function` 21, `make_stability_halting_function` 20, `never_halt` 19b, `passive_diffusion` 11, and `simulate` 67.

# 14 Heterogeneous agents

73 ⟨*src/heterogeneous-sds.py* 73⟩≡

```
import random, functools, sds
from collections import namedtuple

def random_hypothesis_function(random):
    return random.randint(0,100)

def test_is_odd(hypothesis):
    return hypothesis % 2 == 1

def test_is_even(hypothesis):
    return hypothesis % 2 == 0

def test_is_large(hypothesis):
    return random.randint(0,100) < hypothesis

def test_is_small(hypothesis):
    return not test_is_large(hypothesis)

def is_prime(hypothesis):
    return hypothesis in (2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,

def is_square(hypothesis):
    return hypothesis in (1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

TestList = namedtuple('TestList',('last_agent','tests'))

# last_agent refers to the number of the first agent which does not use
# the tests.

microtests = [
    TestList(last_agent=1000,tests=[
        is_square,
        test_is_small,
        test_is_even,
    ]),
    TestList(last_agent=2000,tests=[
        test_is_large,
        is_prime,
    ]),
]

def heterogeneous_test_phase(swarm, microtests, random, multitesting=1, multitest_functi

    def zip_agents_and_tests(swarm, microtests):

        prev_last_agent = 0
```

```
        for last_agent_num, test_list in microtests:

            yield (swarm[prev_last_agent:last_agent_num], test_list)

            prev_last_agent = last_agent_num

    for (
        partial_swarm,
        partial_microtests,
    ) in (
        zip_agents_and_tests(swarm, microtests)
    ):

        for _ in sds.generic_test_phase(
            partial_swarm,
            partial_microtests,
            random,
            multitesting,
            multitest_function,
            scalar,
        ):
            pass


agent_count = 2000
swarm = sds.Agent.initialise(agent_count)

max_iterations = 4000
diffusion_function = sds.context_sensitive_diffusion
#diffusion_function = sds.passive_diffusion

clusters = sds.run(
    swarm,
    microtests,
    random_hypothesis_function,
    max_iterations,
    diffusion_function,
    random,
    test_phase_function=heterogeneous_test_phase,
)

print(clusters.most_common(10))
```

Uses `Agent` 4, `context_sensitive_diffusion` 13, `generic_test_phase` 6, `initialise` 5a, `passive_diffusion` 11, and `run` 30.

# 15 True Restaurant Game

75 ⟨*src/restaurant-game.py* 75⟩≡

```
import random, functools, sds
from collections import namedtuple
from pprint import pprint

cuisine_count = 3

restaurant_count = 10

agent_count = 1000

swarm = sds.Agent.initialise(agent_count)

microtests = None

max_iterations = 10000

diffusion_function = sds.passive_diffusion

def clip(num, smallest, largest):
    return min(largest,max(num, smallest))

def make_restaurant(cuisine_count):

    return sorted(
        [0] + [
            random.random()
            for _
            in range(cuisine_count-1)
        ]
    )

restaurants = [
    make_restaurant(cuisine_count)
    for restaurant_num
    in range(restaurant_count)
]

def to_probabilities(r):
    return [t-p for p,t in zip(r,r[1:] + [1])]

#pprint([to_probabilities(r) for r in restaurants])

def generic_random_hypothesis_function(restaurant_count, random):

    return random.randrange(restaurant_count)
```

```
random_hypothesis_function = functools.partial(
    generic_random_hypothesis_function,
    restaurant_count)

def make_taste(cuisine_count):

    general_happiness = random.gauss(0,1)

    return [
        clip(random.gauss(2/3,0.5)+general_happiness,smallest=0,largest=1)
        for _
        in range(cuisine_count)
    ]

    #taste = [1] * cuisine_count

    #taste[random.randrange(cuisine_count)] = 0

    return taste

tastes = [
    make_taste(agent_count)
    for agent_num
    in range(agent_count)
]

def pick_dish(restaurant):

    num = random.random()

    return sum(1 for x in restaurant if num >= x)

def taste_test(restaurant, agent_tastes):

    dish = pick_dish(restaurant)

    agent_taste = agent_tastes[dish]

    return random.random() < agent_taste

def exhaustive_test(tastes, restaurants):


    restaurant_scores = []

    for r_num, restaurant in enumerate(restaurants):
        probs = to_probabilities(restaurant)
        agent_scores = []
        for taste in tastes:
```

```
            agent_scores.append(
                sum(dish*prob for dish,prob in zip(taste,probs))
            )
        restaurant_score = sum(agent_scores)/len(tastes)
        restaurant_scores.append(
            (r_num, restaurant_score)
        )

    return restaurant_scores

def restaurant_game_test_phase(
    swarm,
    microtests,
    random,
    **kwargs
):

    for agent_num, agent in enumerate(swarm):

        agent_tastes = tastes[agent_num]

        agent.active = taste_test(
            restaurants[agent.hypothesis],
            agent_tastes)

clusters = sds.run(
    swarm,
    microtests,
    random_hypothesis_function,
    max_iterations,
    diffusion_function,
    random,
    report_iterations=1000,
    test_phase_function=restaurant_game_test_phase,
)

print(clusters)

truth = exhaustive_test(tastes, restaurants)

scores = [score for num,score in truth]

for hyp, count in clusters.most_common():
    print("Restaurant {hyp} has {c} agents, and a score of {s}".format(
        hyp=hyp,
        c=count,
        s=scores[hyp],
    ))
```

```
truth.sort(key=lambda x:-x[1])
print(truth[:10])
```

Uses `Agent` 4, `initialise` 5a, `passive_diffusion` 11, and `run` 30.

## 16 Library definition

79 ⟨*src/library/setup.py* 79⟩≡

```
from setuptools import setup

setup(
    name='sds',
    version='0.1.4',
    packages=['sds'],
    description='Stochastic Diffusion Search',
    keywords = ['swarm','artificial','intelligence','search'],
    classifiers = [
        'Development Status :: 3 - Alpha',
        'Intended Audience :: Developers',
        'Intended Audience :: Science/Research',
        'License :: OSI Approved :: Apache Software License',
        'Operating System :: OS Independent',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python',
        'Topic :: Scientific/Engineering :: Artificial Intelligence',
    ],
    url='http://www.aomartin.co.uk/sds-library/',
    author='Andrew Owen Martin',
    author_email='a.martin@gold.ac.uk',
    long_description="""\
A library which implements the main variants of Stochastic Diffusion
Search (SDS), and provides a convenient front end.

Stochastic Diffusion Search (SDS) is a generic population-based search
method. SDS agents perform cheap, partial evaluations of a hypothesis (a
candidate solution to the search problem). They then share information
about hypotheses (diffusion of information) through direct one-to-one
communication. As a result of the diffusion mechanism, high-quality
solutions can be identified from clusters of agents with the same
hypothesis.

This is a library used during the writing of my PhD thesis, I will
publish full documentation and host the code on GitHub once the design
has settled down and I have submitted my thesis. Until then, feel free
to email me.

SDS has a Scholarpedia page:
http://www.scholarpedia.org/article/Stochastic_diffusion_search

A list of papers written on SDS can be found in the Stochastic Diffusion
Search paper repository, maintained by the author of this module:
http://www.aomartin.co.uk/sds-repository/publications.html
"""
)
```

80 ⟨*src/library/sds/__init__.py* 80⟩≡
⟨*library dependencies* 5c⟩
⟨*agent class definition* 4⟩
⟨*agent namedtuple* 5b⟩
⟨*generic test phase function* 6⟩
⟨*test phase function* 8⟩
⟨*scalar test phase function* 9⟩
⟨*generic single agent test function* 10⟩
⟨*generic diffusion function* 16⟩
⟨*generic single agent diffusion function* 17⟩
⟨*passive diffusion function* 11⟩
⟨*context free diffusion function* 12⟩
⟨*context sensitive diffusion function* 13⟩
⟨*synchronous iterate function* 23⟩
⟨*asynchronous iterate function* 25⟩
⟨*parallel iterate function* 27⟩
⟨*parallel update state function* 29⟩
⟨*never halt function* 19b⟩
⟨*stability halt function* 20⟩
⟨*instant threshold halt function* 21⟩
⟨*threshold time halt function* 22⟩
⟨*run function* 30⟩
⟨*run daemon* 34⟩
⟨*coupled diffusion* 36b⟩
⟨*coupled test phase* 37⟩
⟨*synchronous coupled test phase* 40⟩
⟨*sequential coupled test phase* 41⟩
⟨*coupled iterate* 42⟩
⟨*coupled run* 43⟩
⟨*count clusters function* 44a⟩
⟨*write swarm function* 33a⟩
⟨*activity function* 45⟩
⟨*estimate noise* 46⟩
⟨*swarm from clusters* 47⟩
⟨*pretty print clusters with values* 48⟩
⟨*sds simulator* 67⟩
⟨*swarm class* 49⟩
⟨*make multilayer sds function* 55⟩
⟨*single diffusion function* 57a⟩
⟨*random compound hypothesis function* 57b⟩
⟨*flatten hypothesis* 58a⟩

81 ⟨*src/library/LICENSE.txt* 81⟩≡

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
```

the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or

implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

```
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

85a ⟨*src/library/MANIFEST.in* 85a⟩≡

```
include LICENSE.txt
include doc/string_search.py
include doc/documentation.pdf
include doc/tutorial.html
include doc/tutorial.pdf
```

85b ⟨*src/library/README.rst* 85b⟩≡

```
This is the readme for the sds module.
```

85c ⟨*src/library/doc/string-search.py* 85c⟩≡

⟨*src/string-search.py* 61⟩

### 16.1 Deploying to PyPi

I deployed this to PyPi by going into `thesis/src/library` and running `python3 setup.py sdist upload`.

I deployed an update by updating the version line in ⟨*src/library/setup.py* 79⟩, running `make pip_install` and running `python setup.py sdist upload` in the virtual environment.

## 17 Mathematical library definition

85d ⟨*src/mathlib/setup.py* 85d⟩≡

```
from setuptools import find_packages, setup

setup(
    name='sdsmath',
    version='0.1',
    packages=find_packages(),
    url='www.aomartin.co.uk',
    author='Andrew Owen Martin',
    author_email='a.martin@gold.ac.uk',
)
```

86a ⟨*src/mathlib/sdsmath/sdsmath.py* 86a⟩≡
⟨*math library dependencies* 86b⟩
⟨*diffusion rate model* (never defined)⟩
⟨*basic model* (never defined)⟩
⟨*minimum stability model* (never defined)⟩

86b ⟨*math library dependencies* 86b⟩≡ (86a)

```
from sympy import Symbol as _Symbol
from sympy import Eq as _Eq
from sympy import solveset as _solveset
from sympy import latex as _latex
from sympy import simplify as _simplify
```

86c ⟨*src/mathlib/sdsmath/__init__.py* 86c⟩≡

```
from .sdsmath import *
```

86d ⟨*src/mathlib/LICENSE.py* 86d⟩≡

86e ⟨*src/mathlib/MANIFEST.in* 86e⟩≡

86f ⟨*src/mathlib/README.rst* 86f⟩≡

```
This is the readme for the sdsmath module.
```

## 17.1 Interacting Markov Chain Model

This is the interacting markov chain model implemented in Sympy.

87 ⟨*interacting markov chain model* 87⟩≡

```
import sympy

pminus, pm, N = sympy.symbols('p^{-} p_{m} N')

a = 2*(1-pminus)*(1-pm)
pi1 = (a-1 + sympy.sqrt( pow(a-1,2) + 2*a*pm*(1-pminus) ) ) / a
pi2 = (1 - sympy.sqrt( pow(a-1,2) + 2*a*pm*(1-pminus) ) ) / a

n = ((N+1)*pi1-1,(N+1)*pi1)

En = N*pi1

sigma = sympy.sqrt(N * pi1 * pi2)

def interacting_markov_chain_model(pminus_val, pm_val, N_val):
    subs = [(N,N_val),(pm,pm_val),(pminus,pminus_val)]
    ngeq, nleq = (n_eq.subs(subs) for n_eq in n)

    return {
        'n >=':ngeq,
        'n <=':nleq,
        'E[n]':En.subs(subs),
        'sigma':sigma.subs(subs),
    }
```