TokenSPICE: EVM Agent-Based Token Simulator

⚠️ *Note: as of mid 2023, this codebase is not being maintained. It might work, it might not. If you find a bug, feel free to report it, but do not expect it to be fixed. If you do a PR where tests pass, we're happy to merge it. And feel free to fork this repo and change it as you wish (including bug fixes).*

TokenSPICE simulates tokenized ecosystems via an agent-based approach, with EVM in-the-loop.

It can help in Token Engineering flows, to design, tune, and verify tokenized ecosystems. It's young but promising. We welcome you to contribute! 👋

- TokenSPICE simulates by simply running a loop. At each iteration, each *agent* in the *netlist* takes a step. That's it! Simple is good.
- A netlist wires up a collection of agents to interact in a given way. Each agent is a class. It has an Ethereum wallet, and does work to earn money. Agents may be written in pure Python, or with an EVM-based backend.
- One models a system by writing a netlist and tracking metrics (KPIs). One can write their own netlists and agents to simulate whatever they like. The netlists directory has examples.

Contents

🏗 Initial Setup

Prerequisites

- Linux/MacOS
- Python 3.8.5+
- solc 0.8.0+ [Instructions]
- ganache. To install: `npm install ganache --global`
- nvm 16.13.2, *not* nvm 17. To install: `nvm install 16.13.2; nvm use 16.13.2`. [Details]

Install TokenSPICE

Open a new terminal and:

#clone repo

git clone https://github.com/tokenspice/tokenspice

cd tokenspice

#create a virtual environment

python3 -m venv venv

#activate env

source venv/bin/activate

#install dependencies

pip install -r requirements.txt

#install brownie packages (you can ignore FileExistsErrors)

./brownie-install.sh

Potential issues & workarounds

- Issue: Brownie doesn't support Python 3.11 yet. Workaround: before "install dependencies" step above, run `pip install vyper==0.3.7 --ignore-requires-python` and `sudo apt-get install python3.11-dev`
- Issue: MacOS might flag "Unsupported architecture". Workaround: install including ARCHFLAGS: `ARCHFLAGS="-arch x86_64" pip install -r requirements.txt`

Run Ganache

From "Prerequisites", you should have Ganache installed.

Open a new console and go to tokenspice directory. Then:

source venv/bin/activate

./ganache.py

This will start a Ganache chain, and populate 9 accounts.

TokenSPICE CLI

`tsp` is the command-line interface for TokenSPICE.

Open a new console and go to tokenspice directory. Then:

```
source venv/bin/activate



#add pwd to bash path

export PATH=$PATH:.



#see tsp help

tsp
```

Compile the contracts

NOTE: if you have a directory named `contracts` from before, which is side-by-side with your `tokenspice` directory, you'll get [issues](). To avoid this, rename or move that contracts directory.

From the same terminal:

```
#install 3rd party libs, then call "brownie compile" in sol057/ and sol080/

tsp compile
```

TokenSPICE sees smart contracts as classes. How:

- When it starts, it calls `brownie.project.load('./sol057', name="MyProject")` to load the ABIs in `./sol057/build/`. Similar for `sol080`.
- That's enough info to treat each contract in `sol057/contracts/` as a *class*. Then, call `deploy()` on it to create a new *object*.

🏄 Running, Debugging

Testing

From terminal:

```
#run single test. It uses brownie, which auto-starts Ganache local blockchain node.

pytest sol057/contracts/simpletoken/test/test_Simpletoken.py::test_transfer



#run all of a directory's tests

pytest sol057/contracts/simpletoken/test



#run all unit tests

pytest



#run static type-checking. By default, uses config mypy.ini. Note: pytest does dynamic type-checking.

mypy ./



#run linting on code style
```

pylint *

#auto-fix some pylint complaints

black ./

Go here for details on linting / style.

Simulating with TokenSPICE

From terminal:

#run simulation, sending results to 'outdir_csv' (clear dir first, to be sure)

rm -rf outdir_csv; tsp run netlists/scheduler/netlist.py outdir_csv

You'll see an output like:

Arguments: NETLIST=netlists/...

Launching 'ganache-cli --accounts 10 --hardfork ...

mnemonic: 'sausage bunker giant drum ...

INFO:master:Begin.

INFO:master:SimStrategy={OCEAN_funded=5.0, duration_seconds=157680000, ...}

INFO:master:Tick=0 (0.0 h, 0.0 d, 0.0 mo, 0.0 y); timestamp=1642844072; OCEAN_vested=0, ...

INFO:master:Tick=3 (2160.0 h, 90.0 d, 3.0 mo, 0.2 y); timestamp=1650620073; OCEAN_vested=0.0, ...

INFO:master:Tick=6 (4320.0 h, 180.0 d, 6.0 mo, 0.5 y); timestamp=1658396073; OCEAN_vested=0.0, ...

INFO:master:Tick=9 (6480.0 h, 270.0 d, 9.0 mo, 0.7 y); timestamp=1666172074; OCEAN_vested=0.0, ...

INFO:master:Tick=12 (8640.0 h, 360.0 d, 12.0 mo, 1.0 y); timestamp=1673948074; OCEAN_vested=0.0, ...

INFO:master:Tick=15 (10800.0 h, 450.0 d, 15.0 mo, 1.2 y); timestamp=1681724074; OCEAN_vested=0.232876 ...

Now, let's view the results visually. In the same terminal:

#create output plots in 'outdir_png' (clear dir first, to be sure)

rm -rf outdir_png; tsp plot netlists/scheduler/netlist.py outdir_csv outdir_png

#view plots

eog outdir_png

To see the blockchain txs apart from the other logs: open a *new* terminal and:

#activate env't

cd tokenspice

source venv/bin/activate

#run ganache

export PATH=$PATH:.

tsp ganache

Now, from your original terminal:

#run the sim. It will auto-connect to ganache

```
rm -rf outdir_csv; tsp run netlists/scheduler/netlist.py outdir_csv
```

For longer runs (eg wsloop), we can log to a file while watching the console in real-time:

```
#run the sim in the background, logging to out.txt

rm -rf outdir_csv; tsp run netlists/wsloop/netlist.py outdir_csv > out.txt 2>&1 &



#monitor in real-time

tail -f out.txt
```

To kill a sim in the background:

```
#find the background process

ps ax |grep "tsp run"



#example result:

#223429 pts/4    Rl     0:02 python ./tsp run netlists/wsloop/netlist.py outdir_csv



#to kill it:

kill 223429
```

Debugging from Brownie Console

Brownie console is a Python console, with some extra Brownie goodness, so that we can interactively play with Solidity contracts as Python classes, and deployed Solidity contracts as Python objects.

From terminal:

#brownie needs a directory with ./contracts/. Go to one.

cd sol057/

#start console

brownie console

In brownie console:

```
>>> st = Simpletoken.deploy("DT1", "Simpletoken 1", 18, Wei('100 ether'), {'from': accounts[0],
"priority_fee": chain.priority_fee, "max_fee": chain.base_fee + 2 *

 chain.priority_fee})

Transaction sent: 0x9d20d3239d5c8b8a029f037fe573c343efd9361efd4d99307e0f5be7499367ab

  Gas price: 0.0 gwei   Gas limit: 6721975

  Simpletoken.constructor confirmed - Block: 1   Gas used: 601010 (8.94%)

  Simpletoken deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87



>>> st.symbol()

'DT1'



>>> st.balanceOf(accounts[0])/1e18
```

>>> dir(st)

[abi, address, allowance, approve, balance, balanceOf, bytecode, decimals, decode_input, get_method, get_method_object, info, name, selectors, signatures, symbol, topics, totalSupply, transfer, transferFrom, tx]

🦑 Agents and Netlists

Agents Basics

Agents are defined at `agents/`. Agents are in a separate directory than netlists, to facilitate reuse across many netlists.

All agents are written in Python. Some may include EVM behavior (more on this later).

Each Agent has an [`AgentWallet`](), which holds a [`Web3Wallet`](). The `Web3Wallet` holds a private key and creates transactions (txs).

Netlists Basics

The netlist defines what you simulate, and how.

Netlists are defined at `netlists/`. You can reuse existing netlists or create your own.

What A Netlist Definition Must Hold

TokenSPICE expects a netlist module (in a netlist.py file) that defines these specific classes and functions:

- `SimStrategy` class: simulation run parameters
- `KPIs` class and `netlist_createLogData()` function: what metrics to log during the run
- `netlist_plotInstructions()` function: how to plot the metrics after the run

- `SimState` class: system-level structure & parameters, i.e. how agents are instantiated and connected. It imports agents defined in `agents/*Agent.py`. Some agents use EVM. You can add and edit Agents to suit your needs.

How to Implement Netlists

There are two practical ways to specify `SimStrategy`, `KPIs`, and so on for netlist.py:

1. For simple netlists. Have just one file (`netlist.py`) to hold all the code for each class and method given above. This is appropriate for simple netlists, like simplegrant (just Python) and simplepool (Python+EVM).
2. For complex netlists. Have one or more *separate files* for each class and method given above, such as `netlists/NETLISTX/SimStrategy.py`. Then, import them all into `netlist.py` file to unify their scope to a single module (`netlist`). This allows for arbitrary levels of netlist complexity. The wsloop netlist is a good example. It models the Web3 Sustainability Loop, which is inspired by the Amazon flywheel and used by Ocean, Boson and others as their system-level token design.

Agent.takeStep() method

The class `SimState` defines which agents are used. Some agents even spawn other agents. Each agent object is stored in the `SimState.agents` object, a dict with some added querying abilities. Key `SimState` methods to access this object are `addAgent(agent)`, `getAgent(name:str)`, `allAgents()`, and `numAgents()`. `SimStateBase` has details.

Every iteration of the engine make a call to each agent's `takeStep()` method. The implementation of `GrantGivingAgent.takeStep()` is shown below. Lines 26–33 determine whether it should disburse funds on this tick. Lines 35–37 do the disbursal if appropriate. There are no real constraints on how an agent's `takeStep()` is implemented. This which gives great TokenSPICE flexibility in agent-based simulation. For example, it can loop in EVM, like we show later.

```python
    def takeStep(self, state):
        do_disburse = False
        if self._tick_last_disburse is None:
            do_disburse = True
        else:
            n_ticks_since = state.tick - self._tick_last_disburse
            n_s_since = n_ticks_since * state.ss.time_step
            n_s_thr = self._s_between_grants
            do_disburse = (n_s_since >= n_s_thr)

        if do_disburse:
            self._disburseFunds(state)
            self._tick_last_disburse = state.tick
```

Netlist Examples

Here are some existing netlists.

- simplegrant - granter plus receiver, that's all. No EVM.
- simplepool - publisher that periodically creates new pools. EVM.
- scheduler - scheduled vesting from a wallet. EVM.
- wsloop - Web3 Sustainability Loop. No EVM.
- oceanv3 - Ocean Market V3 - initial design. EVM.
- oceanv4 - Ocean Market V4 - solves rug pulls. EVM.

To learn more about how TokenSPICE netlists are structured, we refer you to the simplegrant (pure Python) and simplepool (Python+EVM) netlists, which each have more thorough explainers.

🐡 Backlog

Larger things we'd like to see:

- Higher-level tools that use TokenSPICE, including design entry, verification, design space exploration, and more.
- Improvements to TokenSPICE itself in the form of faster simulation speed, improved UX, and more.

See this board: https://github.com/orgs/tokenspice/projects/1/views/1

Benefits of EVM Agent Simulation

TokenSPICE and other EVM agent simulators have these benefits:

- Faster and less error prone, because the model = the Solidity code. Don't have to port any existing Solidity code into Python, just wrap it. Don't have to write lower-fidelity equations.
- Enables rapid iterations of writing Solidity code -> simulating -> changing Solidity code -> simulating.
- Super high fidelity simulations, since it uses the actual code itself. Enables modeling of design, random and worst-case variables.
- Mental model is general enough to extend to Vyper, LLL, and direct EVM bytecode. Can extend to non-EVM blockchain, and multi-chain scenarios.
- Opportunity for real-time analysis / optimization / etc against *live chains*: grab the latest chain's snapshot into ganache, run a local analysis / optimization etc for a few seconds or minutes, then do transaction(s) on the live chain. This can lead to trading systems, failure monitoring, more.

Resources

Here are further resources.

- TokenSPICE medium posts, starting with "Introducing TokenSPICE"
- Intro to SPICE & TokenSPICE [Gslides - short] [Gslides - long]
- TE for Ocean V3 [GSlides] [video] , TE Community Workshop, Dec 9, 2020
- TE for Ocean V4 [GSlides] [slides] [video] , TE Academy, May 21, 2021

History:

- TokenSPICE was initially built to model the Web3 Sustainability Loop. It's now been generalized to support EVM, on arbitary netlists.

- Most initial work was by [trentmc](#) ([Ocean Protocol](#)); [several more contributors](#) have joined since 👨‍👩‍👦 👨‍👩‍👧‍👦

Art:

How to Model with TokenSPICE EVM Agent Simulation

From a quickstart to deeper understanding of agents and netlists architecture, by example

1. Introduction

TokenSPICE is a tool that simulates tokenized ecosystems via an agent-based approach. It can help in Token Engineering flows, to design, tune, and verify tokenized ecosystems.

This article builds on two previous articles: an introduction to TokenSPICE, and flows for design & verification with inspiration from the SPICE simulator of Electrical Engineering.

***This* article is written for developers and modelers who want to jump right in and start using TokenSPICE, then learn about how to create their own netlists.** It's organized as follows.

- Sections 2–3 are the quickstart sections from TokenSPICE repo's README: initial setup (section 2), and doing simulation and making changes (section 3).

- Section 4 describes the structure of agents and netlists. Recall that a netlist wires up a collection of agents to interact in a given way.
- Sections 5–6 describe two example netlists. The `simplegrant` netlist (section 5) is in **pure Python** with a grant giver agent and grant receiver agent. The `simplepool` netlist (section 6) is a **simple EVM example**, where a publisher agent periodically creates a Balancer pool. While the netlist itself is simple, it uses Balancer V1 “BPool” Solidity code for a full-fidelity model.

2. Initial Setup

(This is a snapshot from TokenSPICE README. The most up-to-date version can always be found there.)

2. 1 Prerequisites

- Linux/MacOS

- Python 3.8.5+

2.2 Set up environment

Open a new terminal and:

```
#clone repo

git clone https://github.com/oceanprotocol/tokenspice.git

cd tokenspice

#create a virtual environment

python3 -m venv venv

#activate env

source venv/bin/activate
```

```
#install dependencies. Install wheel first to avoid errors.

pip install wheel

pip install -r requirements.txt
```

2.3 Get Ganache running

Think of Ganache as local EVM blockchain network, with just one node.

Open a new terminal and:

```
#install Ganache (if you haven't yet)

npm install ganache-cli --global

#activate env't

cd tokenspice
```

```
source venv/bin/activate
```

```
#run ganache.py. It calls ganache cli and fills in many arguments for you.
```

```
./ganache.py
```

2.4 Deploy the smart contracts to ganache

Below, you will deploy smart contracts from Ocean Protocol. Those contracts include an ERC20 datatoken factory, ERC20 template, Balancer pool factory, Balancer pool template, and metadata management. Each contract has a corresponding Python wrapper in the `web3engine` directory. Then, Python agents in `assets/agents` use these wrappers.

You can add your own smart contracts by deploying them to EVM, then adding corresponding Python wrappers and agents to use them.

Let's do this. Open a new terminal and:

```
#Grab the contracts code from main, *OR* (see below)

git clone https://github.com/oceanprotocol/contracts

#OR grab from a branch. Here's Alex's V4 prototype branch

git clone --branch feature/1mm-prototype_alex
https://github.com/oceanprotocol/contracts
```

Then, deploy. In that same terminal:

```
cd contracts

#one-time install

npm i

#compile .sol, deploy to ganache, update
contracts/artifacts/*.json

npm run deploy
```

Finally, open `tokenspice/tokenspice.ini` and set `ARTIFACTS_PATH = contracts/artifacts`.

- Now, TokenSPICE knows where to find each contract on ganache (address.json file)
- And, it knows what each contract's interface is (*.json files).

2.4 Test one EVM-based test

Open a new terminal and:

```
#activate env't

source venv/bin/activate

#run test

pytest web3engine/test/test_btoken.py
```

2.5 First usage of tsp

We use `tsp` for TokenSPICE in the command line.

First, add pwd to bash path. In the terminal:

```
export PATH=$PATH:.
```

To see help, call `tsp` with no args.

```
tsp
```

2.6 Run simulation

Here's an example on a supplied netlist `simplegrant`.

Simulate the netlist, storing results to `outdir_csv`.

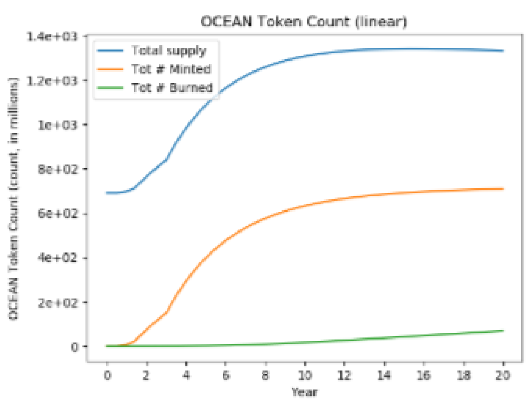
```
tsp run assets/netlists/simplegrant/netlist.py outdir_csv
```

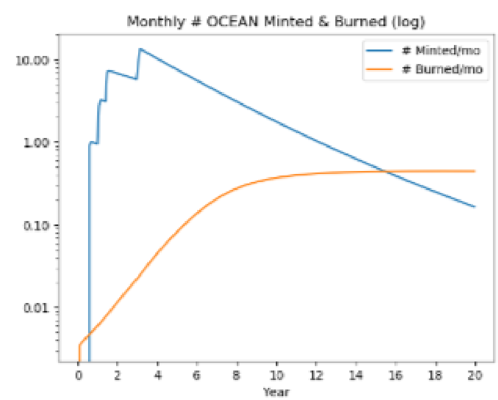
Output plots to `outdir_png`, and view them.

```
tsp plot assets/netlists/simplegrant/netlist.py outdir_csv outdir_png
```

```
eog outdir_png
```

Here are example plots from [wsloop netlist](). They track token count, tokens minted, tokens burned, and tokens granted over a 20 year period.





3. Do Simulations, Make Changes

(This is a snapshot from TokenSPICE [README](). The most up-to-date version can always be found there.)

3.1 Do Once, At Session Start

Start chain. Open a new terminal and:

```
cd ~/code/tokenspice
```

```
source venv/bin/activate
```

```
./ganache.py
```

Deploy contracts. Open a new terminal and:

```
cd ~/code/contracts
```

```
npm run deploy
```

3.2 Do >=1 Times in a Session

Update simulation code. Open a new terminal. In it:

```
cd ~/code/tokenspice
```

```
source venv/bin/activate
```

```
#then use editor to change assets/netlists/foo.py
```

Run tests. In the same terminal as before:

```
#run a single pytest-based test

pytest web3engine/test/test_btoken.py::test_ERC20

#run a single pytest-based test file

pytest web3engine/test/test_btoken.py

#run all tests in util/ directory

pytest util

#run all tests except web3engine/ (slow)

pytest --ignore=web3engine
```

```
#run all tests
```

```
pytest
```

```
#run static type-checking. Dynamic is automatic.
```

```
mypy --config-file mypy.ini ./
```

3.3 Test that everything is working

```
source venv/bin/activate
```

```
pytest
```

Commit changes.

```
git add <changed filename>
```

```
git status -s [[check status]]
```

```
git commit -m <my commit message>
```

```
git push

#or

git status -s [[check status]]

git commit -am <my commit message>

git push
```

4. Agents and Netlists

(This is a snapshot from TokenSPICE README. The most up-to-date version can always be found there.)

4.1 Agents Basics

Agents are defined at `assets/agents/`. Agents are in a separate directory than netlists, to facilitate reuse across many netlists.

All agents are written in Python. Some may include EVM behavior (more on this later).

Each Agent has an `AgentWallet`, which holds a `Web3Wallet`. The `Web3Wallet` holds a private key and creates transactions (txs).

4.2 Netlist Basics

The netlist defines what you simulate, and how.

Netlists are defined at `assets/netlists/`. You can reuse existing netlists or create your own.

4.3 What A Netlist Definition Must Hold

TokenSPICE expects a `netlist` module (in a `netlist.py` file) that defines these *specific* classes and functions:

- `SimStrategy` class: simulation run parameters

- `KPIs` class and `netlist_createLogData()` function: what metrics to log during the run
- `netlist_plotInstructions()` function: how to plot the metrics after the run
- `SimState` class: system-level structure & parameters, i.e. how agents are instantiated and connected. It imports agents defined in `assets/agents/*Agent.py`. Some agents use EVM. You can add and edit Agents to suit your needs.

4.4 How to Implement Netlists

There are two practical ways to specify `SimStrategy`, `KPIs`, and so on for `netlist.py`:

1. **For simple netlists.** Have *just one file* (`netlist.py`) to hold all the code for each class and method given above. This is appropriate for simple netlists, like simplegrant (just Python) and simplepool (Python+EVM).

2. **For complex netlists.** Have one or more *separate files* for each class and method given above, such as `assets/netlists/NETLISTX/SimStrategy.py`. Then, import them all into `netlist.py` file to unify their scope to a single module (`netlist`). This allows for arbitrary levels of netlist complexity. The wsloop netlist is a good example. It models the Web3 Sustainability Loop, which is inspired by the Amazon flywheel and used by Ocean, Boson and others as their system-level token design.

4.5 Agent.takeStep() method

The class `SimState` defines which agents are used. Some agents even spawn other agents. Each agent object is stored in the `SimState.agents` object, a `dict` with some added querying abilities. Key `SimState` methods to access this object are `addAgent(agent)`, `getAgent(name:str)`, `allAgents()`, and `numAgents()`. `SimStateBase` has details.

Every iteration of the engine make a call to each agent's `takeStep()` method. The implementation of [`GrantGivingAgent.takeStep()`]() is shown below. Lines 26–33 determine whether it should disburse funds on this tick. Lines 35–37 do the disbursal if appropriate.

There are no real constraints on how an agent's `takeStep()` is implemented. This which gives great TokenSPICE flexibility in agent-based simulation. For example, it can loop in EVM, like we show later.

```
25      def takeStep(self, state):
26          do_disburse = False
27          if self._tick_last_disburse is None:
28              do_disburse = True
29          else:
30              n_ticks_since = state.tick - self._tick_last_disburse
31              n_s_since = n_ticks_since * state.ss.time_step
32              n_s_thr = self._s_between_grants
33              do_disburse = (n_s_since >= n_s_thr)
```

Implementation of `GrantGivingAgent.takeStep()`

4.6 Netlist Examples

Here are some existing netlists.

- simplegrant — granter plus receiver, that's all. No EVM.
- simplepool — publisher that periodically creates new pools. EVM.
- wsloop — Web3 Sustainability Loop. No EVM.
- (WIP) oceanv3 — Ocean Market V3. Initial design. EVM.
- (WIP) oceanv4 — Ocean Market V4. Solves rug pulls. EVM.

The next two sections will show how TokenSPICE netlists are structured, by elaborating on the simplegrant (pure Python) and simplepool (Python+EVM) netlists.

5. simplegrant Netlist

5.1 Overview

The simplegrant netlist at `assets/netlists/simplegrant/netlist.py` has two agents (objects):

- `granter`, a [GrantGivingAgent]
- `taker`, a [GrantTakingAgent]

As one might expect, `granter` gives grants to `taker` over time according to a simple schedule. This continues until runs out of money. These two agents are instantiated in the netlist's `SimStrategy` class.

Here's the netlist code, in Python. It's just one file that defines `SimStrategy` class, `SimState` class, `KPIs` class, `netlist_createLogData()` function, and `netlist_plotInstructions()` function.

The following subsections elaborate on each of these, sequentially top-to-bottom in the `netlist.py` file. They're worth understanding, because when you create your own netlist, you'll be making your own versions of these.

5.2 Imports

Imports are at the top of the netlist.

```
1  from enforce_typing import enforce_types
2  from typing import List, Set
3
4  from assets.agents import GrantGivingAgent, GrantTakingAgent
5  from engine import KPIsBase, SimStateBase, SimStrategyBase
6  from util.constants import S_PER_HOUR, S_PER_DAY
```

simplegrant: imports

Lines 1–2 import from third-party libraries: `enforce_types` for dynamic type-checking, and `List` and `Set` to specify types for type-checking.

Lines 4–6 import local modules: definitions for grant-giving and grant-taking agents from the `agents` directory; base classes for `KPIs`, `SimState` and `SimStrategy` (more on this later); and some constants.

5.3 SimStrategy Class

The netlist defines the `SimStrategy` class by inheriting from a base class, then injecting code as needed. All magic numbers go here, versus scattering them throughout the code.

```
 9  class SimStrategy(SimStrategyBase.SimStrategyBase):
10      def __init__(self):
11          super().__init__()
12
13          #==baseline
14          self.setTimeStep(S_PER_HOUR)
15          self.setMaxTime(10, 'days')
16
17          #==attributes specific to this netlist
18          self.granter_init_OCEAN: float = 1.0
19          self.granter_s_between_grants: int = S_PER_DAY*3
20          self.granter_n_actions: int = 4
```

simplegrant: SimStrategy

Lines 14–15 define values that every `SimStrategy` needs: a time interval between steps (set to one hour); and a stopping condition (set to 10 days).

Lines 18–20 define values specific to this netlist: how much OCEAN the `granter` starts with (1.0 OCEAN); the time interval between grants (3 days); and the number of grant actions (4 actions, therefore 1.0/4 = 0.25 OCEAN per grant).

5.4 SimState Class

The netlist defines the `SimState` class by inheriting from a base class, then injecting code as needed.

```
23 class SimState(SimStateBase.SimStateBase):
24     def __init__(self, ss=None):
25         assert ss is None
26         super().__init__(ss)
27 
28         #ss is defined in this netlist module
29         self.ss = SimStrategy()
30 
31         #wire up the circuit
32         granter = GrantGivingAgent.GrantGivingAgent(
33             name="granter1",
34             USD=0.0,
35             OCEAN=self.ss.granter_init_OCEAN,
36             receiving_agent_name="taker1",
37             s_between_grants=self.ss.granter_s_between_grants,
38             n_actions=self.ss.granter_n_actions)
39         taker = GrantTakingAgent.GrantTakingAgent(
40             name = "taker1", USD=0.0, OCEAN=0.0)
41         for agent in [granter, taker]:
42             self.agents[agent.name] = agent
43 
44         #kpis is defined in this netlist module
45         self.kpis = KPIs(self.ss.time_step)
```

simplegrant: SimState

Line 28 instantiates an object of class `SimStrategy`. We'd just defined that class earlier in the netlist.

Lines 32–37 instantiates the `granter` object. It's a [`GrantGivingAgent`]() which is defined in `assets/agents`. Like all agent instances, it's given a name ("granter1"), initial USD funds (0.0) and initial OCEAN funds (specified via `SimStrategy`). As a

`GrantGivingAgent`, it needs a few more parameters: the name of the agent receiving funds ("taker1"), the time interval between grants (via `SimStrategy`), and number of actions (via `SimStrategy`). Note how magic numbers are kept out of here; they stay in `SimStrategy`.

Lines 38–39 instantiates the `taker` object. It's a `GrantTakingAgent`. It's given a name ("taker1"); see how this is the same name that the `granter` has specified where funds go. This is how the netlist gets "wired up", similar in philosophy to SPICE. Finally, the `taker`'s initial funds are specified, as 0.0 USD and 0.0 OCEAN.

5.5 KPIs Class

The netlist defines the `KPIs` by inheriting from a base class, then injecting code as needed. The base class already tracks many metrics out-of-the-box including each agent's OCEAN balance at each time step. This netlist doesn't need more, so its `KPIs` class is simply a pass-through.

```
@enforce_types
class KPIs(KPIsBase.KPIsBase):
    pass
```

simplegrant: KPIs

5.6 netlist_createLogData

The netlist defines the `netlist_createLogData()` function, which is called by the core simulator engine [`SimEngine`]() in each `takeStep()` iteration of a simulation run.

```
55 def netlist_createLogData(state):
56     """SimEngine constructor uses this."""
57
58     s = [] #for console logging
59     dataheader = [] # for csv logging: list of string
60     datarow = [] #for csv logging: list of float
61
62     #SimEngine already logs: Tick, Second, Min, Hour, Day, Month, Year
63     #So we log other things...
64
65     g = state.getAgent("granter1")
66     s += ["; granter OCEAN=%s, USD=%s" % (g.OCEAN(), g.USD())]
67     dataheader += ["granter_OCEAN", "granter_USD"]
68     datarow += [g.OCEAN(), g.USD()]
69
70     #done
71     return s, dataheader, datarow
```

simplegrant: netlist_createLogData()

Lines 58–60 initializes these 3 variables: `s`, `dataheader`, and `datarow`. The rest of the routine fills them in, iteratively.

- Line 65 grabs the “granter1” agent from the `SimState` object `state`. The lines below will use that agent. This function can to grab any data from SimState. Since SimState holds all the agents, this function can grab any any agent. The lines that follow query information from the “granter” agent `g`.

- `s` is a list of strings to be logged to the console’s standard output (stdout), where the final string is a concatenation of all items in the list. Line 66 updates `s` with another item, for the `granter`’s OCEAN balance and USD balance. Those values are retrieved by querying the `g` object.
- The other two variables are towards constructing a csv file, where the first row has all the header variable names and each remaining row is another datapoint corresponding to a time step.
- This function constructs `dataheader` as the list of header names. Line 67 adds the “granter_OCEAN” and “granter_USD” variables to that list.

- This function constructs `datarow` as a list of variable values, i.e. a datapoint. Each of these has 1:1 mapping to header names added to `dataheader`. This function queries the the `g` object to fill in the values.

While this netlist (`simplegrant`) only records a single group of values (OCEAN and USD balance), other netlists like [`wsloop`]() record several groups of values.

5.7 netlist_plotInstructions

The netlist defines the `SimState` by inheriting from a base class, then injecting code as needed.

```python
def netlist_plotInstructions(header: List[str], values):
    """
    Describe how to plot the information.
    tsp.do_plot() uses this.

    :param: header: List[str] holding 'Tick', 'Second', ...
    :param: values: 2d array of float [tick_i, valuetype_i]
    :return: x: List[float] -- x-axis info on how to plot
    :return: y_params: List[YParam] -- y-axis info on how to plot
    """
    from util.plotutil import YParam, arrayToFloatList, LINEAR, MULT1, DOLLAR

    x = arrayToFloatList(values[:,header.index("Day")])

    y_params = [
        YParam(["granter_OCEAN"],["OCEAN"],"granter_OCEAN",LINEAR,MULT1,DOLLAR),
        YParam(["granter_USD"],  ["USD"],  "granter_USD",  LINEAR,MULT1,DOLLAR)
    ]

    return (x, y_params)
```

simplegrant: netlist_plotInstructions()

This concludes the description of the `simplegrant` netlist. Further information can be found in [its README](#).

6. simplepool Netlist

6.1 Overview

`simplepool` layers in EVM. Its netlist is at

`assets/netlists/simplepool/netlist.py`

The netlist is a single file, which defines everything needed: `SimStrategy`, `SimState`, etc.

The netlist's `SimState` creates a `PublisherAgent` instance, which during the simulation run creates `PoolAgent` objects.

Each `PoolAgent` holds a full-fidelity EVM Balancer pool as follows:

- At the top level, each `PoolAgent` Python object (an agent) holds a `pool.BPool` Python object (a driver to the lower level).
- One level lower, each `BPool` Python object (a driver) points to a `BPool.sol` contract deployment in Ganache EVM (actual contract).

You can view `pool.BPool` as a middleware driver to the contract deployed to EVM. Like all drivers, is in the `web3engine/` directory.

7. Conclusion

This article targeted developers and modelers wanting to jump right in and start using TokenSPICE, then learn about how to create their own netlists. It reviewed how agents and netlists work, and then used worked examples on two simple netlists — a pure Python one and an EVM-based one.

As a next step, we encourage you to go to the TokenSPICE repo, and go through the README to try it for yourself:)