Tutorial for Smart Contract Development with Python: Application to local Energy markets

The aim of the document is to set up the environment on Windows to compile and run a Smart Contract from Python. Interaction with Matlab is also addressed quickly.

I. About Smart Contract

Smart Contracts are piece of software code that run on a blockchain, and are executed within a virtual environment (stack based virtual machine) defined within the network of nodes that mine and store the blockchain. The virtual environment is shared among all nodes (to be verified). However, the virtual environment is completely isolated and has no access to the rest of the network (file systems). Nodes that execute the code will get reward (gas). The Smart Contract can be used to interact with the blockchain, and to store values, to realize an auction, to act as a bank (receive deposit, withdrawals), and many other applications.
Here, we ultimately want to design a Smart Contract that will receive bids and offers from different agents (buyers and sellers). It will also store money deposited by these agents who have an account on the Blockchain. A blockchain is not necessarily dedicated to crypto-currency. In our case, we will consider the Ethereum Blockchain, and will consider that agents have an ethereum account so that they can exchange money for their energy consumption.
Hence, we can distinguish:

1. The physical network used to store the blockchain. It is constituted of nodes (computers), that mine, and execute codes (for the blockchain or for Smart Contracts)
2. The Blockchain, that stores all the data (crypto-currency transactions and/or other data(?)). Storage is distributed/replicated among all nodes
3. From the Blockchain, we can follow all the transactions in order to determine the list of all the "external" accounts, with their public and private keys, but also the amount of crypto-currency.
4. Finally, we also can list the contract accounts, that are defined by an address, a bytecode abi, and a quantity of ether.

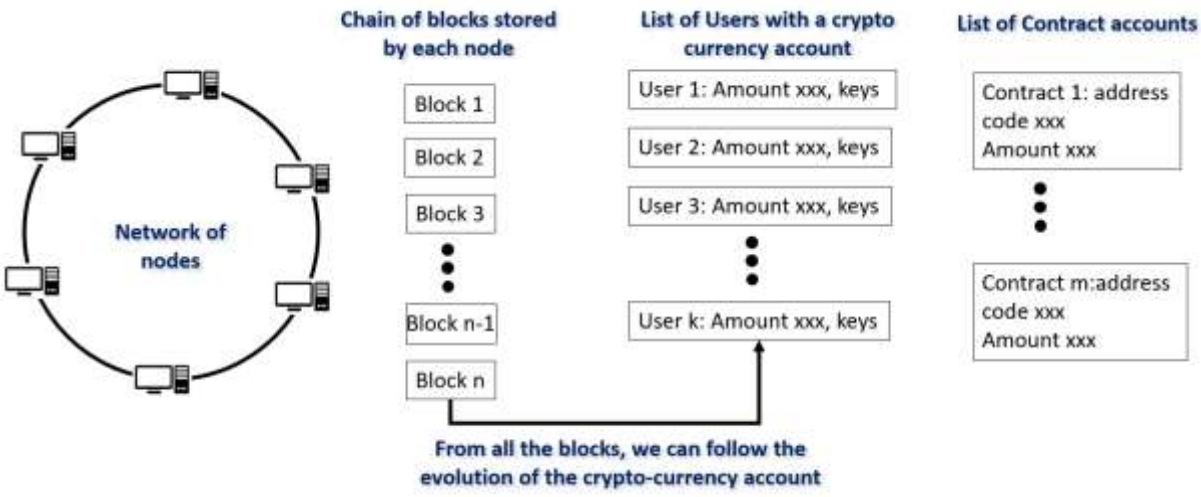

Figure 1. Simplified vision of Blockchain that is used (among other used) for crypto-currency

In this tutorial, we will create a smart contract that we will deploy on a blockchain. We will then run it by interacting with the network. We could deploy the smart contract on the ethereum blockchain, but

this would cost us some money. So instead of deploying on the Ethereum blockchain, we will generate a local blockchain on our computer, with many accounts already defined, with ether allocated.
The architecture of the project would be the following if it was deployed on a real Ethereum blockchain:

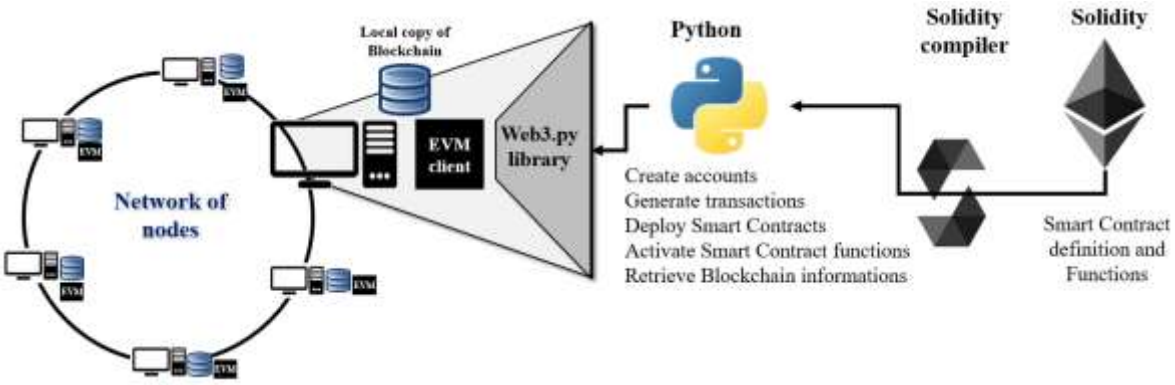

Figure 2. Project architecture as it would be in a real Ethereum deployment

As we will work locally, the actual architecture is the following:

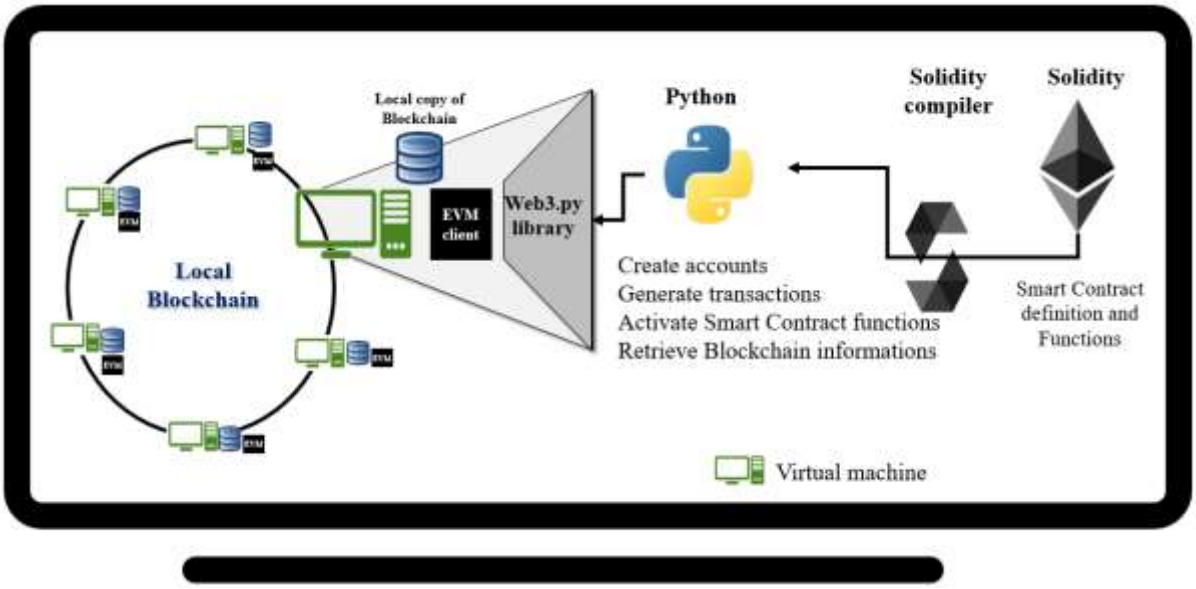

Figure 3. Actual architecture with a local blockchain

The steps are the following:

1. We configure a local blockchain with nodes (virtual machines) and accounts
2. We develop a smart contract in solidity
3. We compile it using solidity compiler (called by python or not)
4. We deploy the compiled code (byte code) in the blockchain using python
5. We interact with the contract (and the blockchain) through python commands that are sent to the address of the smart contract via a node of the local blockchain

In the next section, we will set-up our environment to run a local blockchain and interact with it using Python.

II. Environment Set-up

Most of the content here comes from the youtube channel listed below:

https://www.youtube.com/watch?v=SAi5rYFh7yw&list=PLS5SEs8ZftgVn38FOhXvLc0PoX_0hnJO9&index=1

A. Install the local blockchain

An easy way to configure a local blockchain is to install Ganache, accessible from truffle website: https://www.trufflesuite.com/ganache
Once it is install, you can launch Ganache, and select "Quickstart Ethereum" to generate a local blockchain with 10 accounts already configured with 100 ether on each, as shown below:

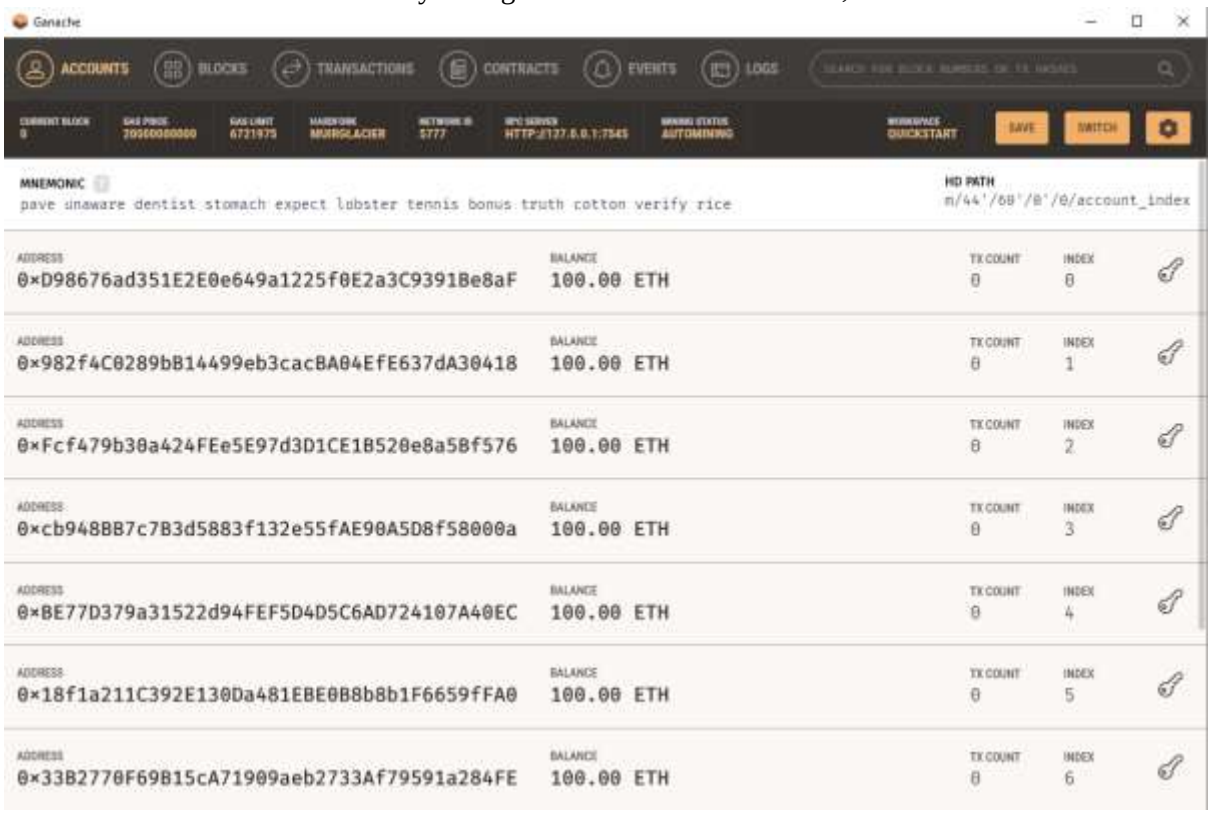

Figure 4. Ganache - Local Blockchain: visualization of every account, with their ether amount and address

B. Install and configure Python

You must install Python. I recommend to install anaconda for Python 3.7 as it includes many packages for AI development, and will also install pip, that will be needed to install the web3 library: https://www.anaconda.com/products/individual
Once anaconda is installed, I personally use VS Code studio to run my Smart Contract applications, as VS Code is able to handle python and Solidity. Hence, I recommend to install VS Code: https://code.visualstudio.com/download
Then, you need to configure VS Code as it is not yet configured to compile Python, nor Solidity code.
Once VS Code is launched, you can install Python's extension as shown below, by clicking on the "extensions" icon on the left, and typing python in the search bar:

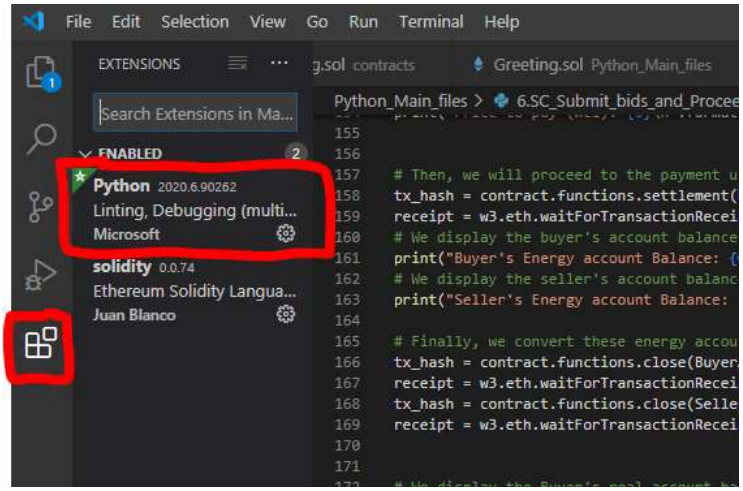

Figure 5. Python Extension for VS Code

C. Install Solidity Compiler

Exactly as for the python extension, in VS Code, look for the solidity extension from Juan Blanco, and install it.

D. Creation of the workspace

To create your workspace, just click file>open Floder> here, create a folder where you want ("*SmartContractVSCode_Tutorial*"). (I am far from a VS Code user, so I am just going directly to what gives the results. And this way, we do not really create a workspace, but well…).
You should have something as shown below:

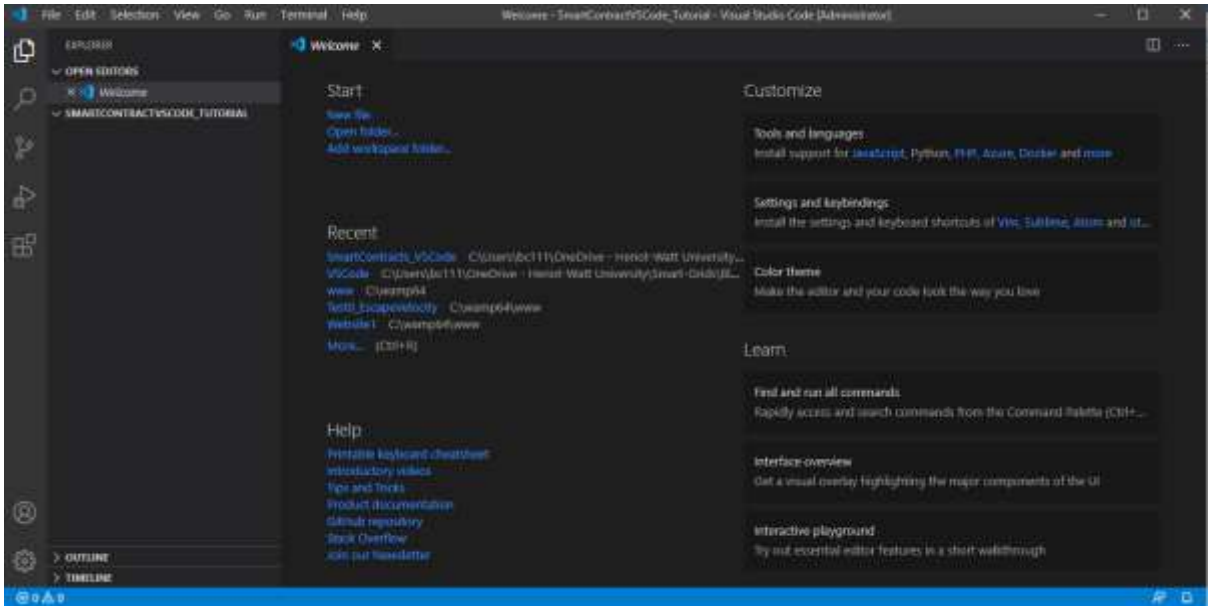

Figure 6. Workspace start in VS Code

We now can create 2 folders, one for the Smart Contracts (solidity codes), one for the Python programs that will interact with the blockchain, as shown below (it can be done by right clicking on the space below "SmartContractVSCode_Tutorial"):

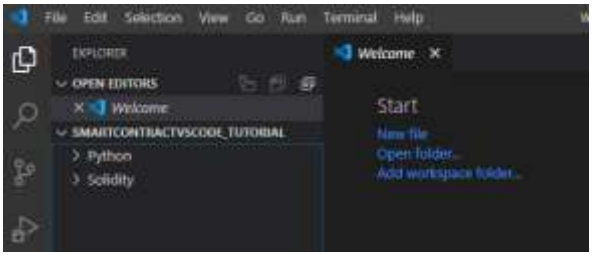

Figure 7. Workspace Architecture

E. Install Web3.py

To interact with the blockchain, we need to install the "web3.py" library in our Python environment. Before doing this, we will now create a first python file, in which we will aim to connect to the local Ganache Blockchain: from VS Code, create a file named "*1_ConnectionGanache.py*" in the Python folder. Then double click on it. Once VS Code has opened it, it will launch the Python extension. You can see which instance of Python has been launched in the bottom bar on the left, as shown below. I recommend to use the Python 3.7 base conda or to generate a virtual environment. If you want to change the instance, simply click on the word "python" in the bottom bar left (circled in red in Figure 8), this will open a list in the top bar, as shown in Figure 9. Select the one linked with Conda.

Then, we need to install web3.py. To do that, make sure the terminal is displayed (otherwise, go to view> Terminal), and in the terminal, type "*pip install web3*".
The installation should be straightforward. If it is not, make sure python (anaconda) has been added to the windows path variables (check "environment variables windows" on google) and is high enough in the environment variables list. You can also do the same from the windows command line (cmd) or from the powershell. You might want to have admin rights for the installation (ie launch the terminal or VS Code with admin rights if you are not the administrator of your PC).

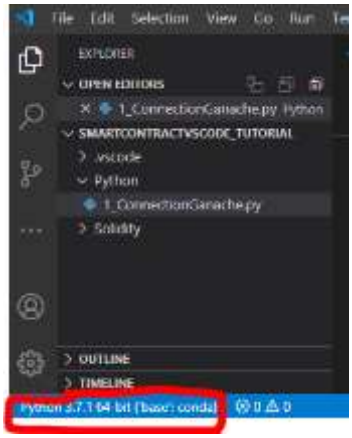

Figure 8. Python instance launched

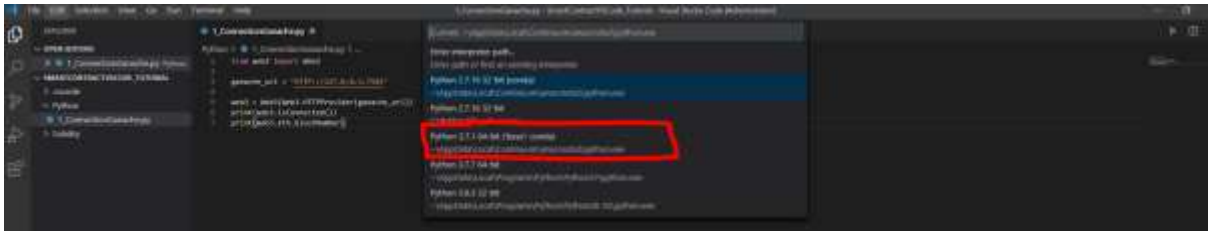

Figure 9. Choice of Python Instance

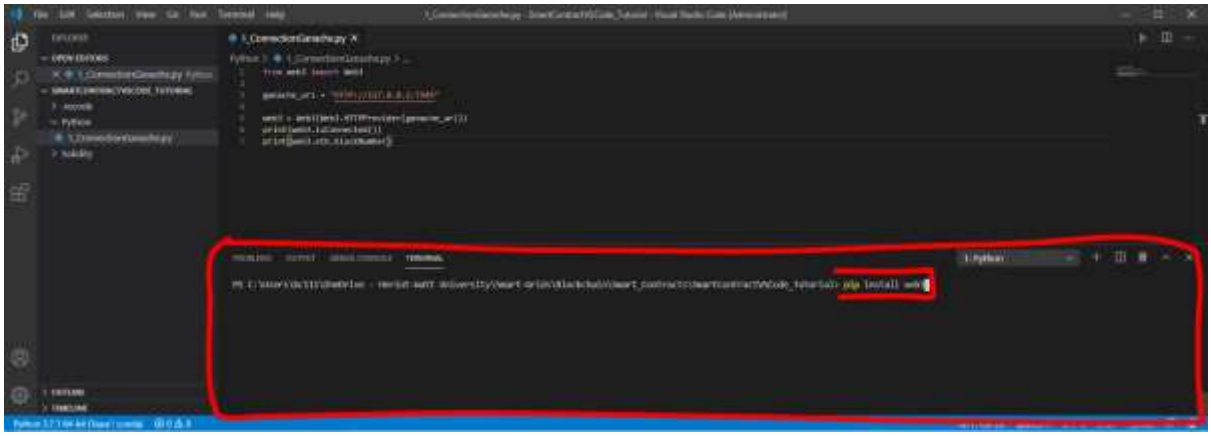

Figure 10. Install web3.py

Now, we can start writing a program to connect to the local Blockchain.
In the editor, type:

```
from web3 import Web3

ganache_url = "HTTP://127.0.0.1:7545"

web3 = Web3(Web3.HTTPProvider(ganache_url))
print(web3.isConnected())
print(web3.eth.blockNumber)
```

Where the ganache_url was taken from the ganache interface, as shown below:

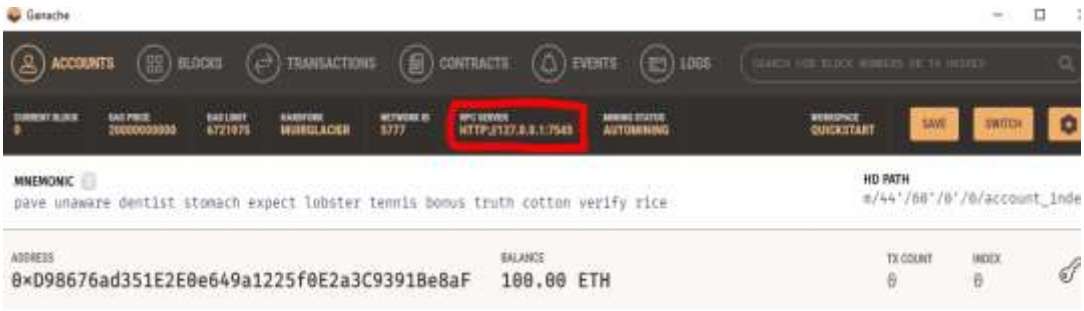

Figure 11. Ganache interface for web3 connection

Make sure Ganache is up and running.
You can then launch the code by pressing ctrl+F5 or clicking on the green arrow in the right part of the upper bar.
Outputs should be:
True
0

You can see that the block number (0) matches what is given by Ganache.
If the code resulted in an error “No Module web3”, whereas the installation went through (you can check that web3 is installed by typing “pip freeze” in the terminal, and find web3 in the list of installed libraries), this means there is an issue between the Python instance that is called and the web3 installation. I have similar issues, that are quite hard to resolve (some have not been resolved yet on my computer, so I do not know how to solve all of them)…

III. Interaction between Python and the local Blockchain

In this section, we will give an example of interaction between Python and the local Blockchain in order to realize transactions in the Blockchain:
Create a second python file called: “2_Transaction_Ganache.py” and copy the following code:

```
# """ Before running this, you need to install Ganache to run a local Blockchain. Otherwise, use infura to access a node. """
# Here, addresses match the local Ganache Blockchain.

from web3 import Web3

ganache_url = "HTTP://127.0.0.1:7545"

w3 = Web3(Web3.HTTPProvider(ganache_url))
print(w3.isConnected())
print(w3.eth.blockNumber)

account_1 = "0xD98676ad351E2E0e649a1225f0E2a3C9391Be8aF"
account_2 = "0xE35ae3bdA7a787EE2ac7f1c50C7483BB3418B9b2"
private_Key_1 = "27fb7dbfbf162bdf7d6f49828d7a79cc79a6ac8f4810030c31a3de611509b594"

nonce = w3.eth.getTransactionCount(account_1)

tx = {
    'nonce': nonce,
    'to': account_2,
    'value': w3.toWei(0.1,'ether'),
    'gas': 2000000,
    'gasPrice': w3.toWei('20','gwei'),
}
signed_tx = w3.eth.account.signTransaction(tx,private_Key_1)
tx_hash = w3.eth.sendRawTransaction(signed_tx.rawTransaction)
print(w3.toHex(tx_hash))

Block1= w3.eth.getBlock(1)
print(Block1)
```

Change the account_1 and account _2 addresses so they match the address that are displayed in your Ganache UI, as shown below (be careful to not have any space when copy pasting):

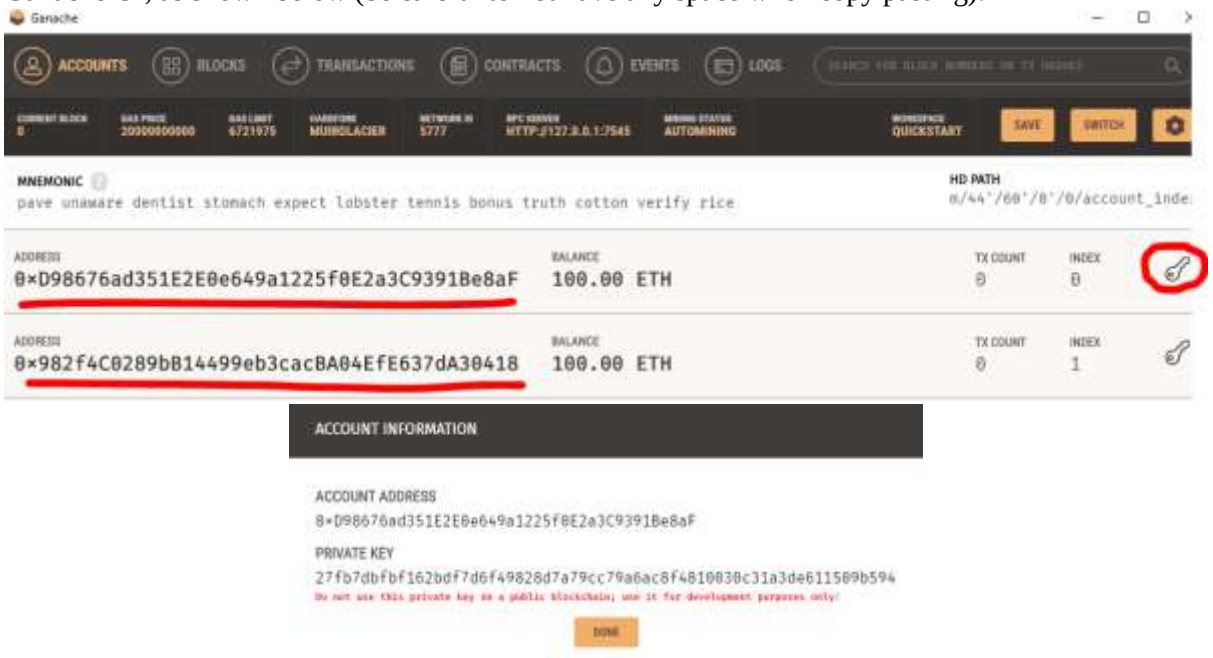

Figure 12. Account informations from Ganache blockchain’s users

The transaction will transfer ether from account_1 to account_2. In the Python terminal, you can see the receipt of the transaction, whereas Ganache updates the account amounts:

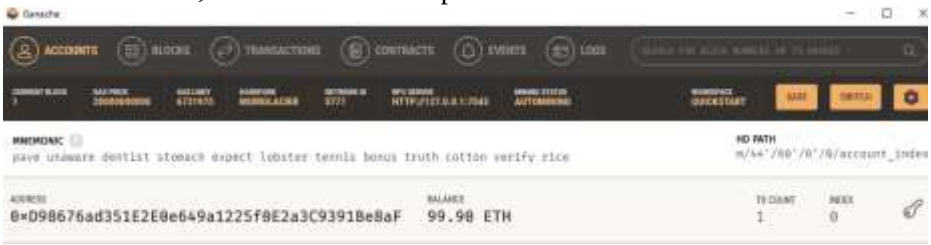

Figure 13. Update of Accounts after the transaction

IV. Smart Contracts Examples

In this section, we will use Smart Contracts to interact with the Blockchain. Hence, we will start to code Smart Contracts using Solidity langage.

A. Hello World

In Solidity, the Hello world program corresponds to a greeting program.
In the "Solidity" folder, create a file "Greeting.sol" in which you can type (solidity codes are in purple in this document):

```
// Use VS Code, install Solidity extension from VS Code (the one from Juan Blanco), and compile with F5.

pragma solidity >=0.4.21 <0.7.0;

contract Greeter {
  string public greeting;

  constructor() public {
    greeting = 'Hello';
  }

  function setGreeting(string memory _greeting) public {
    greeting = _greeting;
  }

  function greet() view public returns (string memory) {
    return greeting;
  }
}
```

To compile the code, in VS Code, press F5. The Terminal should display: "Compilation completed successfully!, with 1 warnings". If compilation did not work, check the solidity compiler in VS Code (google). Indeed, as Solidity evolves very quickly, the code given in this example might be out of date very soon. Also, you can increase the number of solidity compiler version by changing the line "`pragma solidity >=0.4.21 <0.7.0;`" to the highest number of the current compiler.
Finally, if nothing works, you can also use "remix", an online solidity compiler.
The compilation should also create a new folder (bin) as shown below. In this folder, you can find 2 files: Greeter.abi and Greeter.bin. They will be useful as when we deploy the smart contract, we will send their content to the EVM (Ethereum Virtual Machine).

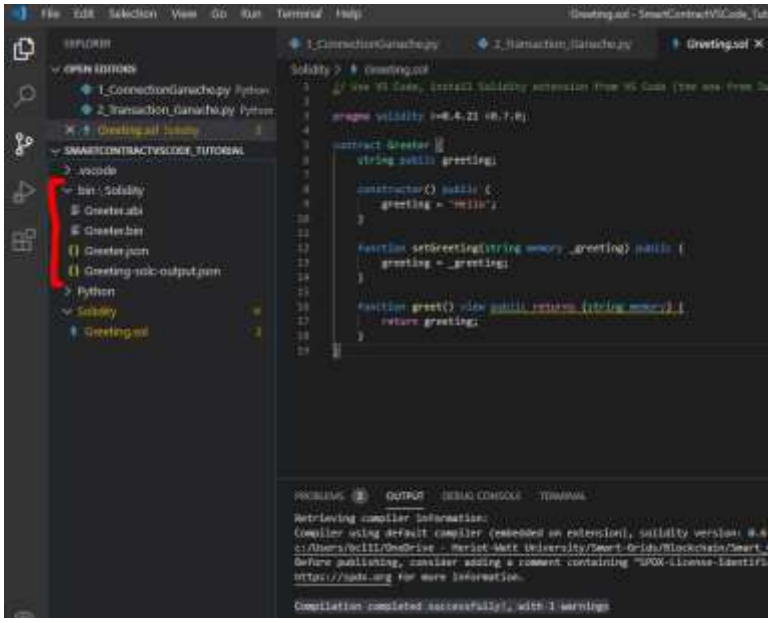

Figure 14. Output from Solidity compilation

Now, we will deploy this contract to the blockchain. To do so, we will use Python (and web3) as a pipeline. In the python folder, create a file "3_SC_Launch_Greet.py" with the following code:

```
# """ Before running this, you need to install Ganache to run a local Blockchain. Otherwise, use infura to access a node. """
# Here, addresses match the local Ganache Blockchain.
# Then, You should compile the Greeting.sol file (solidity Smart Contract) by pressing F5. This will generate a Greeter.json.
# You then need to retrieve the abi and Bytecode of your Smart Contract. To do so:
# 1. the bytecode is generated by the compilation, and can be retrieved in the bin\Greeter.json (find the "bytecode" line)
# 2. the abi: in the window of the Greeter.sol solidity file, on VsCode, press F1, and type in: Solidity: Code generate ... ABI... the abi is
then in the folder bin, the file named Greeter.abi  (btw, the bytecode is in the Greeter.bin file)
import json
from web3 import Web3

ganache_url = "HTTP://127.0.0.1:7545"
w3 = Web3(Web3.HTTPProvider(ganache_url))
print(w3.isConnected())
print(w3.eth.blockNumber)

# set the default account eth.accounts
#web3.eth.defaultAccount = '0xWalletAddress';
w3.eth.defaultAccount = w3.eth.accounts[0]
abi =
json.loads('[{"inputs":[],"stateMutability":"nonpayable","type":"constructor"},{"inputs":[],"name":"greet","outputs":[{"internalType
":"string","name":"","type":"string"}],"stateMutability":"view","type":"function"},{"inputs":[],"name":"greeting","outputs":[{"internal
Type":"string","name":"","type":"string"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"string","name":"_gr
eeting","type":"string"}],"name":"setGreeting","outputs":[],"stateMutability":"nonpayable","type":"function"}]')
bytecode = "608060405234801561001057600080fd5b50604080518"

GreeterSC = w3.eth.contract(abi = abi,bytecode=bytecode)
tx_hash = GreeterSC.constructor().transact()
tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)

contract = w3.eth.contract(
   address = tx_receipt.contractAddress,
   abi = abi
)
print(contract.functions.greet().call())
```

Where you must replace the content of abi and bytecode between the quotes by the content of the Greeter.abi and Greeter.bin files respectively.

When you launch the python file (green arrow), it should give:
True
1
Hello

B. Hello world with Python compilation

In this section, we will do the same thing, but the compilation will be directly called by Python.
In order to do so, we must install solidity compiler for python.
To do so, in the terminal, type: “pip install solc”
However, it might not be enough, so I would recommend to install also solcx: "pip install py-solc-x"
Once everything is installed, you can create a file in the python folder, named “4_SC_Compile_and_Launch_Greet.py” and copy the following code, where you must change the paths that are highlighted in red so they point to your solidity file “Greeting.sol”:

```
# """ Before running this, you need to install Ganache to run a local Blockchain. Otherwise, use infura to access a node. """
# Here, addresses match the local Ganache Blockchain.
# Then, You should (maybe) install solc (open a powershell terminal or a cmd, and run "pip install solc")
# Then, You should (definitely) install solcx (open a powershell terminal or a cmd, and run "pip install py-solc-x")
https://pypi.org/project/py-solc-x/
# Finally, you should change the paths that are listed below to locate the solidity SC (here, Greeting.sol).
# import json
import time
import pprint
from web3 import Web3
from solcx import compile_source, compile_files
from solcx import install_solc
install_solc('v0.5.3')

#Python methods for interactions with Contracts : https://web3py.readthedocs.io/en/stable/contracts.html#contract-functions
# Interactions with Contracts example: https://web3py.readthedocs.io/en/stable/contracts.html
# Examples of Solidity contracts: https://solidity.readthedocs.io/en/v0.5.10/solidity-by-example.html


def compile_source_file(file_path):
  with open(file_path, 'r') as f:
    source = f.read()

  return compile_source(source)


def deploy_contract(w3, contract_interface):
   tx_hash = w3.eth.contract(
      abi=contract_interface['abi'],
      bytecode=contract_interface['bin']).constructor().transact()
#functions.transact() executes the specified function by sending a new public transaction.
   address = w3.eth.getTransactionReceipt(tx_hash)['contractAddress']
   return address


# Connection to he Local Ganache Blockchain
ganache_url = "HTTP://127.0.0.1:7545"
w3 = Web3(Web3.HTTPProvider(ganache_url))
print(w3.isConnected())
print(w3.eth.blockNumber)
```

```
w3.eth.defaultAccount = w3.eth.accounts[0]


#open('c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/Greeting.sol', 'r')


contract_source_path = 'c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/Greeting.sol'
compiled_sol = compile_source_file('c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/Greeting.sol')
contract_id, contract_interface = compiled_sol.popitem()

#print(contract_interface)
abi = contract_interface['abi']
bytecode = contract_interface['bin']
print(abi)
# Deployment of the contract
address = deploy_contract(w3, contract_interface)
""" GreeterSC = w3.eth.contract(abi = abi,bytecode=bytecode)
tx_hash = GreeterSC.constructor().transact()
tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash) """
contract = w3.eth.contract(
   address = address,
   abi = abi
)
#Display the result
print(contract.functions.greet().call())
print("Deployed {0} to: {1}\n".format(contract_id, address))
gas_estimate = contract.functions.setGreeting("Bonjour !").estimateGas() #setGreeting is the function defined in the Greeting.sol
smart contract
print("Gas estimate to transact with setGreeting: {0}\n".format(gas_estimate))

if gas_estimate < 100000:
  print("Sending transaction SetGreeting(Bonjour)\n")
  tx_hash = contract.functions.setGreeting("Bonjour !!").transact()
#functions.transact() executes the specified function by sending a new public transaction.
  receipt = w3.eth.waitForTransactionReceipt(tx_hash)
  print("Transaction receipt mined: \n")
  pprint.pprint(dict(receipt))
  print("Was transaction successful? \n")
  pprint.pprint(receipt['status'])
else:
  print("Gas cost exceeds 100000")

tx_hash = contract.functions.greet().transact()
#functions.transact() executes the specified function by sending a new public transaction.
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
# we check that the greeting has been updated ton "bonjour"
print(contract.functions.greet().call())
#functions.transact() calls a contract function, executing the transaction locally using the eth_call API. This will not create a new
public transaction.
```

If everything was well installed, running the code should:

- Connect to the local blockchain
- Give the current number of blocks
- Print the abi of the smart contract
- Print the address of the Smart Contract deployed
- Use the “setGreeting” function from the Smart contract to change the greeting to “Bonjour !!”
- Display the gas estimation
- Display the receipt from asking the blockchain to run the SetGreeting function
- Display the new greeting (Bonjour !!) by calling the other function “greet()” from the Smart contract

Result is shown below:

```
True
2
[{'constant': False, 'inputs': [{'name': '_greeting', 'type': 'string'}], 'name': 'setGreeting', 'outputs': [], 'payable': False, 'stateMutability': 'nonpayable', 'type': 'function'}, {'constant': True, 'inputs': [], 'name': 'greet', 'outputs': [{'name': '', 'type': 'string'}], 'payable': False, 'stateMutability': 'view', 'type': 'function'}, {'constant': True, 'inputs': [], 'name': 'greeting', 'outputs': [{'name': '', 'type': 'string'}], 'payable': False, 'stateMutability': 'view', 'type': 'function'}, {'inputs': [], 'payable': False, 'stateMutability': 'nonpayable', 'type': 'constructor'}]
Hello
Deployed <stdin>:Greeter to: 0x4F6c8079E578990BC94fff6b2c63CccFdec03c66

Gas estimate to transact with setGreeting: 30528

Sending transaction SetGreeting(Bonjour)

Transaction receipt mined:

{'blockHash': HexBytes('0x8c42b9a80e4c4e9e267b115c55a21f465c62aa6744eb8985c3d15bd5b9362862'),
 'blockNumber': 4,
 'contractAddress': None,
 'cumulativeGasUsed': 29138,
 'from': '0xD98676ad351E2E0e649a1225f0E2a3C9391Be8aF',
 'gasUsed': 29138,
 'logs': [],
 'logsBloom':
HexBytes('0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000'),
 'status': 1,
 'to': '0x4F6c8079E578990BC94fff6b2c63CccFdec03c66',
 'transactionHash': HexBytes('0x21db95bf52226f8b0d95eb55489bda7c672711652fe4140df6c5d099b1ca4727'),
 'transactionIndex': 0}
Was transaction successful?

1
Bonjour !!
```

V. Smart Contract for Local Energy Markets

In this final section, we will create a local energy market place in which 3 agents (an operator/DSO, one buyer and one seller) will interact to exchange energy and payment.

A first version of contract will allow the agents to submit bids, and the dso/operator to retrieve the bids, so he can verify offline if the bids are valid.

The second version of the contract will expand the use case to proceed to the payment between the 2 agents once the DSO has validated the energy exchange.

The whole concept is the following:

1. The blockchain exist. In our case, this is a local blockchain
2. Every agent (DSO, Buyers and sellers) has an account in the Blockchain (Ethereum account)
3. In the python file, we compile the solidity Smart Contract
4. The DSO deploys the contract in the local blockchain
5. The DSO makes a deposit of ether in order to run some transactions that will be done by the Smart Contract (could perhaps be avoided)
6. Then Agents (buyers and sellers) submit bids by calling the SC function submitBid(). Bids consist of:
    a. The price per MWh (integer)
    b. The energy quantity in kWh (integer)
    c. Their weights (for negotiation) (integer also)
    d. The type of the bid (1 for buying energy or 0 for selling)

    Meanwhile, when submitting the bid, the buyer agents must transfer money (ether. By giving argument to the .transact() function when calling the submitBid() function) that will be stored in the Smart Contract to their virtual Energy Account.

7. The DSO retrieves all the bids one by one by calling the SC function retrievebid(AgentAddress). So the DSO must know beforehand the addresses of all the agents. We could change that by having an iterative mapping in the Smart Contract, but the easiest way to do is the one proposed here.
8. Then, the DSO/operator computes the negotiation between the 2 agents (offline, as we agreed). The DSO/operator might notify the agents of the outcome of the negotiation. Then, he will wait for the energy transaction to happen. Once it is done, the DSO will determine the price to pay (we could have a quantity of energy and a penalty rate maybe. But it is not useful in this configuration). Note that the confirmation that the energy quantity was delivered comes from the DSO as he owns the smart meter. All this is done in a Matlab file (or Python). In which case we also need to link Python to Matlab.
9. Once the DSO has determined the settlement (price to pay, after the energy was effectively traded), the DSO calls the SC settlement(buyer, seller, amount) function, that will process the money transfer between the buyer and the seller, by using the money that was deposited by the buyer when he submitted the bid.
10. Once everything is finished (all agents have been paid), the DSO closes the marketplace by calling the SC close(agent) function for each agent (not efficient, but this is the easiest way to do it as long as we do not have iterative mapping). This will reimburse the money remaining in the energy account to each agent after the energy was paid.
11. Closing of the SC and reimbursement of the remaining operational deposit from the DSO

Steps 1 to 7 (included) are covered by the subsection A, while subsection B implements all the steps.

A. Bids/offers submission

Create the solidity file: “SC_0_Bid_and_Retrieve.sol” and copy the following code:

```
// This Smart Contract opens a negotiation place (Market Place): First, it will collect the bids from buyers and sellers. It will store the bids.
// Then, it will proceed to the paymen of the selected bids, where the selection is done outside the Smart Contract, by an external agent (DSO)

pragma solidity >=0.4.21 <0.7.0;

contract NegotiationMarketPlace {
  // The Market place stores the count of all the buyers, sellers who sent bids
  uint8 private buyerCount;
  uint8 private sellerCount;
  // The market place stores (with a hash) the bid (price and quantity, we also need the weights (or Beta) and other informations to be added),
and the type of the agent (Buyer (1) or Seller (0))
  mapping (address => uint) private bidPrice;
  mapping (address => uint) private bidQuantity;
  mapping (address => uint) private bidWeight;
  mapping (address => uint) public agentType;
  // The market place is owned by the DSO Agent:
  address payable public dsoAgent;
  // We store all agent's addresses in a list
  address[] public agentsList;
  // We also need to add a time: Times are either absolute unix timestamps (seconds since 1970-01-01) or time periods in seconds.
  uint public marketEndTime; // not used yet

  // Log the event about a bid being made by an address (Agent) and its amount
  event LogBidMade(address indexed accountAddress, uint price, uint quantity, uint Weight); // Notice that weights are uint for now, as fixed
point are not yet fully implemented (=> the DSO will divide by 10)

  // We need  a boolean to state the stage of the market place (open or not for receiving new offers/bids) Set to true at the end, disallows any
change. By default initialized to `false`.
  bool ended;

  //  We create a modifier that could be used later to restrict some functions to only the operator. not needed for simple use
  modifier onlyOperator() {
    require(
      msg.sender == dsoAgent,
      "Only DSO/Operator can call this."
    );
    _;
```

```
    }

    // Constructor function to initialize the MarketPlace. It is "payable" so it can receive initial funding to cover up some mispayment
    constructor() public payable {
      // require(msg.value == 10 ether, "10 ether initial funding required");
       /* Set the owner to the creator of this contract */
       dsoAgent = msg.sender;
       // initialization of variables
       buyerCount = 0;
       sellerCount = 0;
    }

    /// Creation of a bid from an agent return The type of the agent (1 or 0)
    function submitBid(uint _bidprice, uint _bidquantity, uint _bidweight, uint _agenttype) public returns (uint) {
       if (_agenttype == 1) {
          buyerCount++;}
       else {
          sellerCount++;
       }
       // We update all values of this agent
       bidPrice[msg.sender] = _bidprice;
       bidQuantity[msg.sender] = _bidquantity;
       bidWeight[msg.sender] = _bidweight;
       agentType[msg.sender] = _agenttype;

       return agentType[msg.sender];
    }

  // The operator or DSO wants to retrieve all the bids. However, mapping does not allow to do that. Hence, we can either store each bid/agent into an iterable mapping
  // as shown here https://medium.com/rayonprotocol/creating-a-smart-contract-having-iterable-mapping-9b117a461115 or also the github (https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol)
  // or the operator/DSO makes as many requests to this Smart Contract to retrieve all the bids from all the registered agents. And if an agent has not placed a bid, it returns 0
    function retrievebid(address agentaddress) onlyOperator public view returns(uint[4] memory) {
       uint[4] memory array;
       if (bidPrice[agentaddress] != 0){
          array = [bidPrice[agentaddress], bidQuantity[agentaddress],bidWeight[agentaddress],agentType[agentaddress]];
       }
          return array;
    }

  // After negotiations, the DSO/operator wants to activate payment between agents, while specifying the amount
  }
```

Create a python file named “5.SC_Submit_and_Retrieve_bids.py” and copy the following code, while making sure you update the path to the “SC_0_Bid_and_Retrieve.sol” file (highlighted in red):

```
# """ Before running this, you need to install Ganache to run a local Blockchain. Otherwise, use infura to access a node. """
# Here, addresses match the local Ganache Blockchain.
# Then, You should (maybe) install solc (open a powershell terminal or a cmd, and run "pip install solc")
# Then, You should (definitely) install solcx (open a powershell terminal or a cmd, and run "pip install py-solc-x")  https://pypi.org/project/py-solc-x/
# Finally, you should change the paths that are listed below to locate the solidity SC (here, Greeting.sol).
# import json
import time
import pprint
from web3 import Web3
from solcx import compile_source, compile_files
from solcx import install_solc
install_solc('v0.5.3')


########################## To Run the interaction with Matlab ############################
# First, follow the instructions below: https://uk.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html
# 1. Make sure you have Python in your PATH.
# 2. Find the Matlab root folder. You can use the matlabroot command within Matlab to find it.
# 3. In the windows Command line ("cmd" opened with admin rights) Go to the Matlab root folder in the command line by typing cd "matlabroot\extern\engines\python" (In Windows)
# 4. Type in python setup.py install
# 6. In matlab (opened with admin rights ) type in :cd (fullfile(matlabroot,'extern','engines','python'))
#                                   system('python setup.py install')
# Then, uncomment the 2 following lines:
""" import matlab.engine
matlab_eng = matlab.engine.start_matlab() """
```

```
# This Smart Contract aims to:
# 1. Open a Market Place as defined in SC_1_Bid_and_Payment.sol
# 2. Submit bids from 2 agents (1 seller 1 buyer)
# 3. retrieve the bids (by the DSO/operator)


def compile_source_file(file_path):
  with open(file_path, 'r') as f:
    source = f.read()

  return compile_source(source)


def deploy_contract(w3, contract_interface):
    tx_hash = w3.eth.contract(
      abi=contract_interface['abi'],
      bytecode=contract_interface['bin']).constructor().transact()
#functions.transact() executes the specified function by sending a new public transaction.
    address = w3.eth.getTransactionReceipt(tx_hash)['contractAddress']
    return address

# Connection to he Local Ganache Blockchain
ganache_url = "HTTP://127.0.0.1:7545"
w3 = Web3(Web3.HTTPProvider(ganache_url))
print(w3.isConnected())
print(w3.eth.blockNumber)

# We define the agent's accounts
# The operator/DSO:
w3.eth.defaultAccount = w3.eth.accounts[0]
# The Seller:
SellerAgentAccount = w3.eth.accounts[1]
# The Buyer:
BuyerAgentAccount = w3.eth.accounts[2]

# Compile the contract
contract_source_path = 'c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/SC_0_Bid_and_Retrieve.sol'
compiled_sol = compile_source_file('c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/SC_0_Bid_and_Retrieve.sol')
contract_id, contract_interface = compiled_sol.popitem()

# retrieve the compilation results (abi and bytecode)
abi = contract_interface['abi']
bytecode = contract_interface['bin']
# print(abi)
# Deployment of the contract
address = deploy_contract(w3, contract_interface)
contract = w3.eth.contract(
    address = address,
    abi = abi
)

print("Deployed {0} to: {1}\n".format(contract_id, address))
gas_estimate = contract.functions.submitBid(10, 10, 1, 0).estimateGas() #submitbid the function defined in the SC_1_Bid_and_Payment.sol
smart contract -
# arg(1) =10 is the bid price
# arg(2) =10 is the bidQuantity
# arg(3) =1 is the bidWeight
# arg(4) =0 is the agent type (0 for seller, 1 for buyer)
print("Gas estimate to transact with submitBid: {0}\n".format(gas_estimate))

if gas_estimate < 200000:
    # The seller submits his offer:
  print("Submitting Offer from Seller\n")
  tx_hash = contract.functions.submitBid(10, 10, 1, 0).transact({'from': SellerAgentAccount}) # the
  #functions.transact() executes the specified function by sending a new public transaction.
  receipt = w3.eth.waitForTransactionReceipt(tx_hash)
# arg(1) =10 is the bid price
# arg(2) =10 is the bidQuantity
# arg(3) =1 is the bidWeight
# arg(4) =0 is the agent type  (0 for seller, 1 for buyer)
  # The buyer submits his bid
  tx_hash = contract.functions.submitBid(1, 5, 1, 1).transact({'from': BuyerAgentAccount})
  receipt = w3.eth.waitForTransactionReceipt(tx_hash)
  """   print("Transaction receipt mined: \n")
  pprint.pprint(dict(receipt))
  print("Was transaction successful? \n")
```

```
 pprint.pprint(receipt['status']) """
else:
 print("Gas cost exceeds 200000")

# Now the Operator/DSO retrieves the bids
# First retrieves the seller's bid
tx_hash = contract.functions.retrievebid(SellerAgentAccount).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
sellerData = contract.functions.retrievebid(SellerAgentAccount).call()
# Then retrieves the Buyer's bid
tx_hash = contract.functions.retrievebid(BuyerAgentAccount).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
buyerData = contract.functions.retrievebid(BuyerAgentAccount).call()

# we check that the DSO retrieve the buyer's
print(contract.functions.retrievebid(BuyerAgentAccount).call())

# Then, the DSO/Operator realizes the negotiations and validation of the grid offline (Matlab code)
# We send the values to Matlab, and retrieve the price to pay.
Price_to_pay = 1
# #If using Matlab, please uncomment the following line
""" Price_to_pay = matlab_eng.Compute_Price_to_Pay(buyerData,sellerData)"""
print(Price_to_pay)

# Then, we will proceed to the payment using the Smart Contract
```

Launching the code will:

- Affect the DSO to the first account of the blockchain
- Affect the seller to the 2nd account of the blockchain
- Affect the buyer to the 3rd account of the blockchain
- Deploy the Smart Contract
- Buyer and Seller Send one bid/offer each
- The DSO retrieves the bids
- The DSO computes the price the buyer have to pay

This last point can be done by a matlab file (that would implement the automated negotiation and the power flow + measurement from Smart meter in real life operations). To run the matlab files, it is necessary to link python and matlab by following the instructions below: https://uk.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html

1. Make sure you have Python in your PATH.
2. Find the Matlab root folder. You can use the matlabroot command within Matlab to find it.
3. In the windows Command line ("cmd" opened with admin rights) Go to the Matlab root folder in the command line by typing cd "matlabroot\extern\engines\python" (In Windows)
4. Type in python setup.py install
5. In matlab (opened with admin rights) type in :cd (fullfile(matlabroot,'extern','engines','python'))
   system('python setup.py install')

Then, in the python code, after the imports and before the code starts, use:

```
import matlab.engine
matlab_eng = matlab.engine.start_matlab()
```

Then, in the call, after the DSO retrieved the bids, you can call the matlab function by using:

```
Price_to_pay = matlab_eng.Compute_Price_to_Pay(buyerData,sellerData)
```

In which case you must have a matlab function defined, called “Compute_Price_to_Pay”, and stored in a file “Compute_Price_to_Pay.m” in the Python folder. A very simple code for this matlab function could be:

```
function a = Compute_Price_to_Pay(buyerarray,sellerarray)
       a = mean(buyerarray(1),sellerarray(2))*min(buyerarray(2)*sellerarray(2));
end
```

Where the price = average between the buyer and seller's price, and the energy quantity is the min between the requirement from the buyer and what the seller can produce. The price to pay is the multiplication between the two.
Of course, real automated negotiation are much more complex, and no power flow is implemented here, but this is only for testing.
Note that this part could not be tested, as my Python environment was not compatible with Matlab, so I haven't tested that…
Hence, all the matlab pieces have been commented, and the price to pay is fixed to 1 ether, hard coded in the python file. But ultimately, it should be generated by the matlab negotiation+powerflow (from the bids retrieved by the DSO).

Running the python code should result in the following:

True
0
Deployed <stdin>:NegotiationMarketPlace to: 0x7F003145DA3067f155667a7f05B9db47704dBBdd

Gas estimate to transact with submitBid: 106573

Submitting Offer from Seller

[1, 5, 1, 1] (offer from the buyer)
1

B. Local Marketplace Smart Contract

In this subsection, we have extended the previous smart contract so it allows agents to pay.
The solidity file named "SC_1_Bid_and_Payment.sol" is:

```
// This Smart Contract opens a negotiation place (Market Place): First, it will collect the bids from buyers and sellers. It will store the bids.
// Then, it will proceed to the paymen of the selected bids, where the selection is done outside the Smart Contract, by an external agent (DSO)

pragma solidity >=0.4.21 <0.7.0;

contract NegotiationMarketPlace {
  // The Market place stores the count of all the buyers, sellers who sent bids
  uint8 private buyerCount;
  uint8 private sellerCount;
  uint private operationalDeposit; // an amount of money used to pay all operation fees
  // The market place stores (with a hash) the bid (price and quantity, we also need the weights (or Beta) and other informations to be added),
and the type of the agent (Buyer (1) or Seller (0))
  mapping (address => uint) private bidPrice;
  mapping (address => uint) private bidQuantity;
  mapping (address => uint) private bidWeight;
  mapping (address => uint) public agentType;
  mapping (address => uint) private depositAmount;

  //If we use state machine to coordinate all steps
  enum State { BID_DEPOSIT_OPEN, BID_DEPOSIT_OVER, AWAITING_DELIVERY, COMPLETE, FINISHED}
  State public currState;

  // The market place is owned by the DSO Agent:
  address payable public dsoAgent;
  // We also need to add a time: Times are either absolute unix timestamps (seconds since 1970-01-01) or time periods in seconds.
  uint public marketEndTime; // not used yet

  // Log the event about a bid being made by an address (Agent) and its amount
  event LogBidMade(address indexed accountAddress, uint price, uint quantity, uint Weight); // Notice that weights
  // are uint for now, as fixed point are not yet fully implemented (=> the DSO will divide by 10)

    // We need a boolean to state the stage of the market place (open or not for receiving new offers/bids) Set to true at the end, disallows
any change. By default initialized to `false`.
```

```
    bool ended;

    //  We create a modifier that could be used later to restrict some functions to only the operator. not needed for simple use
    modifier onlyOperator() {
        require(
            msg.sender == dsoAgent,
            "Only DSO/Operator can call this."
        );
        _;
    }

    // Constructor function to initialize the MarketPlace. It is "payable" so it can receive initial funding to cover up some mispayment
    constructor() public payable {
       // require(msg.value == 10 ether, "10 ether initial funding required");
        /* Set the owner to the creator of this contract */
        dsoAgent = msg.sender;
        // initialization of variables
        buyerCount = 0;
        sellerCount = 0;
    }


    // used to refill the amount of the operational deposit
    function operationaldeposit() public payable onlyOperator returns (uint) {
        operationalDeposit = operationalDeposit+msg.value;
         return 1;
    }

    ///  Creation of a bid from an agent return The type of the agent (1 or 0)
    function submitBid(uint _bidprice, uint _bidquantity, uint _bidweight, uint _agenttype) public payable  returns (uint) {
         // if we use state machine, require(currState == State.BID_DEPOSIT_OPEN, "Cannot confirm Bid deposit");
       if (_agenttype == 1) {
            buyerCount++;}
        else  {
            sellerCount++;
        }
        // We update all values of this agent
        bidPrice[msg.sender] = _bidprice;
        bidQuantity[msg.sender] = _bidquantity;
        bidWeight[msg.sender] = _bidweight;
        agentType[msg.sender] = _agenttype;
        depositAmount[msg.sender] = msg.value;
         // if (timesup ***** or agent count reached) currState = State.BID_DEPOSIT_OVER;

         return agentType[msg.sender];
    }

   function getbalance(address agentaddress) public view returns(uint) {
        uint out;
        out = depositAmount[agentaddress];
        return out;
    }
   function getrealbalance(address agentaddress) public view returns(uint) {
        uint out;
            if (agentaddress!=dsoAgent){
                out = agentaddress.balance;
            }
            else
            {
                out = dsoAgent.balance;
            }
        return out;
    }

// The operator or DSO wants to retrieve all the bids. However, mapping does not allow to do that. Hence, we can either store each bid/agent into an iterable mapping
// as shown here https://medium.com/rayonprotocol/creating-a-smart-contract-having-iterable-mapping-9b117a461115 or also the github (https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol)
// or the operator/DSO makes as many requests to this Smart Contract to retrieve all the bids from all the registered agents. And if an agent has not placed a bid, it returns 0
    function retrievebid(address agentaddress) public view returns(uint[5] memory) {  // we store the
    // return into memory, not storage, as we do not ned it outside of the function
        require(msg.sender == dsoAgent,"Only DSO/Operator can call this.");
        // if we use state machine, require(currState == State.BID_DEPOSIT_OVER, "Cannot confirm Bid deposit");
        uint[5] memory array;
        if (bidPrice[agentaddress] != 0){
            array                                                =                                                [bidPrice[agentaddress],
bidQuantity[agentaddress],bidWeight[agentaddress],agentType[agentaddress],depositAmount[agentaddress]];
```

```
        }
            return array;
         //currState = State.AWAITING_DELIVERY;

    }

// After negotiations, the DSO/operator wants to activate payment between agents, while specifying the amount
  function settlement(address buyeraddress, address payable selleraddress, uint pricetopay) external onlyOperator {
        // if we use state machine, require(currState == State.AWAITING_DELIVERY, "Cannot confirm delivery");
        if (depositAmount[buyeraddress]>pricetopay){
            // selleraddress.transfer(pricetopay); // we transfer the money to the seller
            depositAmount[selleraddress] = depositAmount[selleraddress]+pricetopay; // we update the amounts in this SC
            depositAmount[buyeraddress] = depositAmount[buyeraddress] - pricetopay;// we update the amounts in this SC
        }
        //currState = State.COMPLETE;
    }

// Once everything is finished, the DSO/operator close the negotiation by redistributing the money that is in the accounts
  function close(address payable agentaddress) external payable onlyOperator {
        // if we use state machine, require(currState == State.COMPLETE, "Cannot confirm delivery");
            // selleraddress.transfer(depositAmount[selleraddress]); // we transfer the money to the seller
            if (agentaddress!=dsoAgent){
            agentaddress.transfer(depositAmount[agentaddress]); // we transfer the money to the seller
            }
            else
            {
               dsoAgent.transfer(operationalDeposit);
            }
        //currState = State.FINISHED;
    }

    // Once everything is finished, the DSO/operator close the negotiation by redistributing the money that is in the accounts
  function closecontract() external payable onlyOperator {
        // if we use state machine, require(currState == State.COMPLETE, "Cannot confirm delivery");
            // selleraddress.transfer(depositAmount[selleraddress]); // we transfer the money to the seller
            uint fee;
            fee = 1000000000000000000;  // we define is a fee of operational cost that should be removed...
               msg.sender.transfer(operationalDeposit-fee);
        //currState = State.FINISHED;
    }


}
```

Then, the associated python file named “6.SC_Submit_bids_and_ProceedPayment.py “ to deploy and interact with the Smart Contract:

```
# """ Before running this, you need to install Ganache to run a local Blockchain. Otherwise, use infura to access a node. """
# Here, addresses match the local Ganache Blockchain.
# Then, You should (maybe) install solc (open a powershell terminal or a cmd, and run "pip install solc")
# Then, You should (definitely) install solcx (open a powershell terminal or a cmd, and run "pip install py-solc-x")  https://pypi.org/project/py-solc-x/
# Finally, you should change the paths that are listed below to locate the solidity SC (here, Greeting.sol).
# import json
import time
import pprint
from web3 import Web3
from solcx import compile_source, compile_files
from solcx import install_solc
install_solc('v0.5.3')


########################## To Run the interaction with Matlab ############################
# First, follow the instructions below: https://uk.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html
# 1. Make sure you have Python in your PATH.
# 2. Find the Matlab root folder. You can use the matlabroot command within Matlab to find it.
# 3. In the windows Command line ("cmd" opened with admin rights) Go to the Matlab root folder in the command line by typing cd "matlabroot\extern\engines\python" (In Windows)
# 4. Type in python setup.py install
# 6. In matlab (opened with admin rights ) type in :cd (fullfile(matlabroot,'extern','engines','python'))
#                                   system('python setup.py install')
# Then, uncomment the 2 following lines:
""" import matlab.engine
matlab_eng = matlab.engine.start_matlab() """

# This Smart Contract aims to:
# 1. Open a Market Place as defined in SC_1_Bid_and_Payment.sol
```

```
# 2. Submit bids from 2 agents (1 seller 1 buyer)
# 3. retrieve the bids (by the DSO/operator)


def compile_source_file(file_path):
   with open(file_path, 'r') as f:
      source = f.read()

   return compile_source(source)


def deploy_contract(w3, contract_interface):
    tx_hash = w3.eth.contract(
        abi=contract_interface['abi'],
        bytecode=contract_interface['bin']).constructor().transact()
#functions.transact() executes the specified function by sending a new public transaction.
    address = w3.eth.getTransactionReceipt(tx_hash)['contractAddress']
    return address

ether = 10**18 # 1 ether = 1000000000000000000 wei


# Connection to he Local Ganache Blockchain
ganache_url = "HTTP://127.0.0.1:7545"
w3 = Web3(Web3.HTTPProvider(ganache_url))
print(w3.isConnected())
print(w3.eth.blockNumber)

# We define the agent's accounts
# The operator/DSO:
w3.eth.defaultAccount = w3.eth.accounts[0]
DsoAgentAccount = w3.eth.defaultAccount
# The Seller:
SellerAgentAccount = w3.eth.accounts[1]
# The Buyer:
BuyerAgentAccount = w3.eth.accounts[2]



# Compile the contract
contract_source_path = 'c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/SC_1_Bid_and_Payment.sol'
compiled_sol = compile_source_file('c:/Users/bc111/OneDrive - Heriot-Watt University/Smart-
Grids/Blockchain/Smart_Contracts/SmartContracts_VSCode/Python_Main_files/SC_1_Bid_and_Payment.sol')
contract_id, contract_interface = compiled_sol.popitem()

# retrieve the compilation results (abi and bytecode)
abi = contract_interface['abi']
bytecode = contract_interface['bin']
# print(abi)
# Deployment of the contract
address = deploy_contract(w3, contract_interface)
contract = w3.eth.contract(
    address = address,
    abi = abi
)

print("Deployed {0} to: {1}\n".format(contract_id, address))

tx = {
     'from': DsoAgentAccount,
     'to': address,
     'value': w3.toWei(5,'ether'),
     'gas': 2000000,
     'gasPrice': w3.toWei('20','gwei'),
}
tx_hash = contract.functions.operationaldeposit().transact(tx)
receipt = w3.eth.waitForTransactionReceipt(tx_hash)

gas_estimate = contract.functions.submitBid(10, 10, 1, 0).estimateGas() #submitbid the function defined in the SC_1_Bid_and_Payment.sol
smart contract -
# arg(1) =10 is the bid price
# arg(2) =10 is the bidQuantity
# arg(3) =1 is the bidWeight
# arg(4) =0 is the agent type (0 for seller, 1 for buyer)
print("Gas estimate to transact with submitBid: {0}\n".format(gas_estimate))

if gas_estimate < 200000:
```

```
    # The seller submits his offer:
  print("Submitting Offer from Seller\n")
  tx_hash = contract.functions.submitBid(10, 10, 1, 0).transact({'from': SellerAgentAccount}) # the
  #functions.transact() executes the specified function by sending a new public transaction.
  receipt = w3.eth.waitForTransactionReceipt(tx_hash)
# arg(1) =10 is the bid price
# arg(2) =10 is the bidQuantity
# arg(3) =1 is the bidWeight
# arg(4) =0 is the agent type  (0 for seller, 1 for buyer)
  # The buyer submits his bid
  tx = {
      'from': BuyerAgentAccount,
      'to': address,
      'value': w3.toWei(2.9,'ether'),
      'gas': 2000000,
      'gasPrice': w3.toWei('20','gwei'),
}
  tx_hash = contract.functions.submitBid(1, 5, 1, 1).transact(tx)
  receipt = w3.eth.waitForTransactionReceipt(tx_hash)
  print(receipt)
  """   print("Transaction receipt mined: \n")
  pprint.pprint(dict(receipt))
  print("Was transaction successful? \n")
  pprint.pprint(receipt['status']) """
else:
  print("Gas cost exceeds 200000")

# Now the Operator/DSO retrieves the bids
# First retrieves the seller's bid
tx_hash = contract.functions.retrievebid(SellerAgentAccount).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
sellerData = contract.functions.retrievebid(SellerAgentAccount).call()
# Then retrieves the Buyer's bid. It also includes the amount of wei (ether) that are currently deposed in the account, so it cannot be
overpassed.
tx_hash = contract.functions.retrievebid(BuyerAgentAccount).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
buyerData = contract.functions.retrievebid(BuyerAgentAccount).call()
print("ok")
# we check that the DSO retrieve the buyer's
print("Buyer's bid: {0}\n".format(contract.functions.retrievebid(BuyerAgentAccount).call()))
# We display the buyer's account balance:
print("Buyer's account Balance: {0}\n".format(contract.functions.getbalance(BuyerAgentAccount).call()))
# We display the seller's account balance:
print("Seller's account Balance: {0}\n".format(contract.functions.getbalance(SellerAgentAccount).call()))


# Then, the DSO/Operator realizes the negotiations and validation of the grid offline (Matlab code)
# We send the values to Matlab, and retrieve the price to pay.
Price_to_pay = 1*ether
# #If using Matlab, please uncomment the following line
""" Price_to_pay = matlab_eng.Compute_Price_to_Pay(buyerData,sellerData)"""
print("Price to pay (Wei): {0}\n".format(Price_to_pay))


# Then, we will proceed to the payment using the Smart Contract
tx_hash = contract.functions.settlement(BuyerAgentAccount ,SellerAgentAccount,Price_to_pay).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
# We display the buyer's account balance:
print("Buyer's Energy account Balance: {0}\n".format(contract.functions.getbalance(BuyerAgentAccount).call()))
# We display the seller's account balance:
print("Seller's Energy account Balance: {0}\n".format(contract.functions.getbalance(SellerAgentAccount).call()))

# Finally, we convert these energy account into real ether by closing everything and redistributing the money to all the agents
tx_hash = contract.functions.close(BuyerAgentAccount).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
tx_hash = contract.functions.close(SellerAgentAccount).transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)


# We display the Buyer's real account balance:
print("Real Buyer's account Balance (wei): {0}\n".format(contract.functions.getrealbalance(BuyerAgentAccount).call()))
# We display the seller's real account balance:
print("Real Seller's account Balance (wei): {0}\n".format(contract.functions.getrealbalance(SellerAgentAccount).call()))

# The DSO closes the negotiation by taking back the remaining amount from the operational depsoit
tx_hash = contract.functions.closecontract().transact()
receipt = w3.eth.waitForTransactionReceipt(tx_hash)
# We display the Dso's real account balance:
```

```
# print("Real DSO's account Balance (wei): {0}\n".format(w3.eth.getBalance(DsoAgentAccount)))
print("over")
```

Change the paths to match your solidity files, and then run the code.
Results in the terminal should be:

True
5
Deployed <stdin>:NegotiationMarketPlace to: 0xEf5ae5e5C831CB2dbA9109c213BB02c0E76Dc19A

Gas estimate to transact with submitBid: 107473

Submitting Offer from Seller

AttributeDict({'transactionHash': HexBytes('0xf680749541abbab9ab0daa5e0e1a4307e6d8468fe556234a01ea57f37b86eca3'), 'transactionIndex': 0, 'blockHash': HexBytes('0x357caf0bd117da483bad9b1a1d305c38b19b810e74f332a5c5b4b4700908f576'), 'blockNumber': 9, 'from': '0x3Ddc228A836414F5E60c18645ca695Bbf079C07f', 'to': '0xEf5ae5e5C831CB2dbA9109c213BB02c0E76Dc19A', 'gasUsed': 130271, 'cumulativeGasUsed': 130271, 'contractAddress': None, 'logs': [], 'status': 1, 'logsBloom': HexBytes('0x00')})
ok
Buyer's bid: [1, 5, 1, 1, 2900000000000000000]

Buyer's account Balance: 2900000000000000000

Seller's account Balance: 0

Price to pay (Wei): 1000000000000000000

Buyer's Energy account Balance: 1900000000000000000

Seller's Energy account Balance: 1000000000000000000

Real Buyer's account Balance (wei): 98995191160000000000

Real Seller's account Balance (wei): 100995740040000000000

Over

Then, looking at the ganache UI, we can see the new balances:

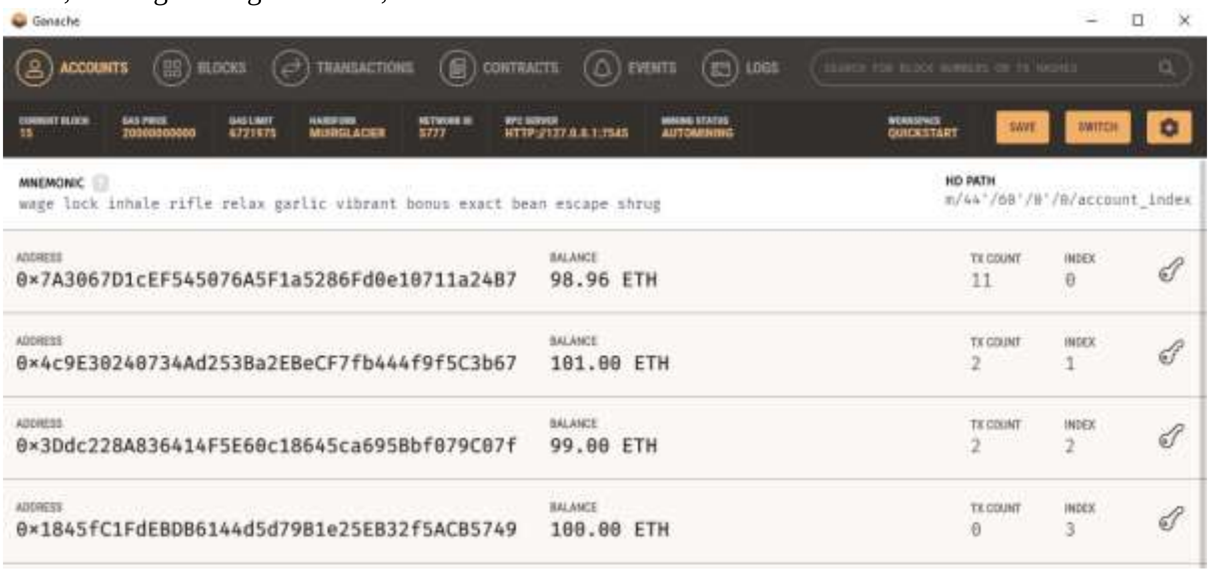

Figure 15. Accounts updated after energy transaction and payment

We can see that the buyer's account (number 3) has lost 1ether, whereas the seller has earned 1 ether. The DSO has less than 100 ether as he did a first deposit to make sure the Smart contract has enough ether to process all the tasks. Note that the DSO account's amount should be corrected in the future.
Future work should also include signature of all transactions, as it was done in the section III with the functions signTransaction() and sendRawTransaction() (no real added value nor complexity. Just requires the private keys).