SIAG/FME Code Quest 2023

DeFi & RoboAdvising Challenge

QuantHub Team

Daniele Maria Di Nosse[1], Federico Gatta[1]

[1] Scuola Normale Superiore, Pisa, Italy

March 14, 2024

Summary

1. Introduction

2. Theoretical Background: The SLSQP + KRR Approach

3. Technical Details

4. Conclusion

Introduction

Introduction

- Decentralized Finance and Crypto markets are hot topics in the financial world.
- Billions of dollars are traded in decentralized exchanges whose behaviour is driven by special programs called Automated Market Makers (AMM).

Introduction

- Decentralized Finance and Crypto markets are hot topics in the financial world.
- Billions of dollars are traded in decentralized exchanges whose behaviour is driven by special programs called Automated Market Makers (AMM).

SIAG/FME Code Quest 2023 tasks

1. Completing the Python **amm class** for reproducing the essential AMM operations
2. Minimizing the Conditional Value at Risk (CVaR) of a liquidity provision strategy.

Introduction

- Decentralized Finance and Crypto markets are hot topics in the financial world.
- Billions of dollars are traded in decentralized exchanges whose behaviour is driven by special programs called Automated Market Makers (AMM).

SIAG/FME Code Quest 2023 tasks

1. Completing the Python **amm class** for reproducing the essential AMM operations
2. Minimizing the Conditional Value at Risk (CVaR) of a liquidity provision strategy.

Notation and Definitions

- We consider $n = 6$ pools and the control variable is the initial wealth distribution: $\boldsymbol{\theta} \in \mathcal{S} = \left\{[0,1]^6 | \sum_{i=1}^{6}(\theta_i) = 1\right\}$. The return distribution at time T can be viewed as $r_T = r_T(\boldsymbol{\theta})$.
- Given a confidence level $\alpha \in [0,1]$, the CVaR is defined as:

$$\text{CVaR}_\alpha(r_T) = \frac{1}{1-\alpha}\int_\alpha^1 \text{VaR}_s(r_T)ds \approx \mathbb{E}[-r_T | -r_T \geq VaR_\alpha]$$

- The target function to minimize is $CVaR_\alpha(\boldsymbol{\theta}) : \boldsymbol{\theta} \rightarrow CVaR_\alpha(r_T(\boldsymbol{\theta}))$. Its evaluation is expensive due to the simulations of 1000 paths for obtaining the empirical $r_T(\boldsymbol{\theta})$ distribution.

Theoretical Background: The SLSQP + KRR Approach

Direct Optimization (only)

First Idea: Direct Optimization

The first idea is to roughly minimize the CVaR function by using a well-known algorithm.

Direct Optimization (only)

First Idea: Direct Optimization

The first idea is to roughly minimize the CVaR function by using a well-known algorithm.

Sequential Least Squares Programming (SLSQP)

Iterative algorithm for non-linear optimization. At each step, the original problem is approximated by a quadratic programming one. The latter is solved via the Newton method applied to the KKT conditions.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} + \boldsymbol{d}_k \tag{1}$$

$$\boldsymbol{d}_k = \underset{\boldsymbol{d}}{\operatorname{argmin}}\ CVaR_\alpha(\boldsymbol{\theta}^{(k)}) + \nabla CVaR_\alpha(\boldsymbol{\theta}^{(k)})^T \boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T \nabla^2_{\theta\theta}\mathcal{L}\left(\boldsymbol{\theta}^{(k)}, \boldsymbol{\lambda}^{(k)}, \sigma^{(k)}\right)\boldsymbol{d} \tag{2}$$

where $\mathcal{L}$ is the Lagrangian function associated to the problem and $\boldsymbol{\lambda}$ and σ are the Lagrange multipliers associated with the inequality and equality constraints.

Direct Optimization (only)

First Idea: Direct Optimization

The first idea is to roughly minimize the CVaR function by using a well-known algorithm.

Sequential Least Squares Programming (SLSQP)

Iterative algorithm for non-linear optimization. At each step, the original problem is approximated by a quadratic programming one. The latter is solved via the Newton method applied to the KKT conditions.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} + \boldsymbol{d}_k \tag{1}$$

$$\boldsymbol{d}_k = \underset{\boldsymbol{d}}{\operatorname{argmin}}\ CVaR_\alpha(\boldsymbol{\theta}^{(k)}) + \nabla CVaR_\alpha(\boldsymbol{\theta}^{(k)})^T \boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T \nabla^2_{\theta\theta}\mathcal{L}\left(\boldsymbol{\theta}^{(k)}, \boldsymbol{\lambda}^{(k)}, \sigma^{(k)}\right)\boldsymbol{d} \tag{2}$$

where $\mathcal{L}$ is the Lagrangian function associated to the problem and $\boldsymbol{\lambda}$ and σ are the Lagrange multipliers associated with the inequality and equality constraints.

Main Pros

- Can handle both linear and non-linear, equality and inequality constraints.
- No assumptions about convexity of the objective function.

Optimization of the Approximated Function

The main SLSQP drawback is related to the computational time, as it performs several sequential and expensive function evaluations.

Optimization of the Approximated Function

The main SLSQP drawback is related to the computational time, as it performs several sequential and expensive function evaluations.

Idea: Approximate the Target Function

We want to approximate the target map $\boldsymbol{\theta} \in \mathcal{S} \to CVaR_\alpha(\boldsymbol{\theta})$ with a cheaper function. Then, the minimization is performed on the approximated function.

Optimization of the Approximated Function

The main SLSQP drawback is related to the computational time, as it performs several sequential and expensive function evaluations.

Idea: Approximate the Target Function

We want to approximate the target map $\boldsymbol{\theta} \in \mathcal{S} \to CVaR_\alpha(\boldsymbol{\theta})$ with a cheaper function. Then, the minimization is performed on the approximated function.

To achieve this goal, we need:

- A dataset of observations $\left\{ \left(\boldsymbol{\theta}^{(i)}, CVaR_\alpha^{(i)} \right) \right\}_{i=1}^{N}$ (generated in parallel).
- A model to learn the target function.

Optimization of the Approximated Function

The main SLSQP drawback is related to the computational time, as it performs several sequential and expensive function evaluations.

Idea: Approximate the Target Function

We want to approximate the target map $\boldsymbol{\theta} \in \mathcal{S} \to CVaR_\alpha(\boldsymbol{\theta})$ with a cheaper function. Then, the minimization is performed on the approximated function.

To achieve this goal, we need:

- A dataset of observations $\left\{ \left(\boldsymbol{\theta}^{(i)}, CVaR_\alpha^{(i)} \right) \right\}_{i=1}^{N}$ (generated in parallel).
- A model to learn the target function.

KRR

It is a Ridge regression (linear regression with $l2$ regularization) where the regressors are obtained by applying a kernel (additive χ^2) transformation, namely:

$$f(\boldsymbol{\theta}) = \sum_{i=1}^{N} \alpha_i K(\boldsymbol{\theta}, \boldsymbol{\theta}^{(i)}) \qquad K(\boldsymbol{\theta}, \boldsymbol{\theta}^{(i)}) = -\sum_{j=1}^{6} \frac{\left(\theta_j - \theta_j^{(i)}\right)^2}{\theta_j + \theta_j^{(i)}} \tag{3}$$

In this setting, with **just N=10 training points**, KRR obtains R^2 score ≈ 0.996.

Two Steps Approach

The KRR approximated minimization has a big drawback: there is no guarantee the argmin $\hat{\boldsymbol{\theta}}_{app}$ for the approximated function coincides with the argmin $\hat{\boldsymbol{\theta}}$ for the target function.

Two Steps Approach

The KRR approximated minimization has a big drawback: there is no guarantee the argmin $\hat{\boldsymbol{\theta}}_{app}$ for the approximated function coincides with the argmin $\hat{\boldsymbol{\theta}}$ for the target function.

Ultimate Idea: Optimization in Two Steps

As standard in the optimization literature, we merge the two approaches previously described by using the faster method to achieve a coarse approximation of the solution. Then, this approximation is used as the starting point for the finer and more expensive method.

Two Steps Approach

Ultimate Idea: Optimization in Two Steps

As standard in the optimization literature, we merge the two approaches previously described by using the faster method to achieve a coarse approximation of the solution. Then, this approximation is used as the starting point for the finer and more expensive method.

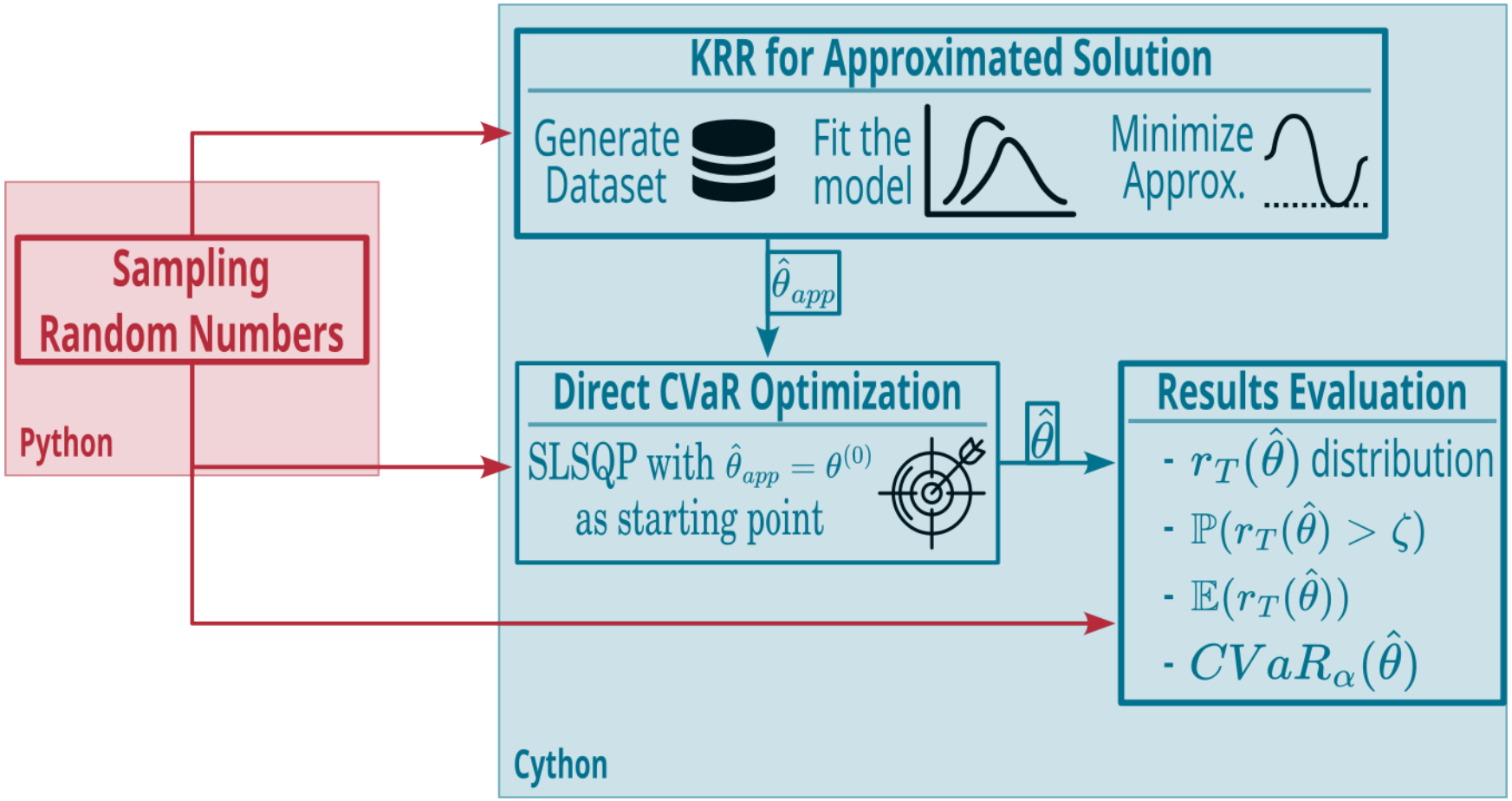
	Grid Search	**KRR**	**SLSQP**	**KRR+SLSQP**
$\mathbb{P}[r_T > \zeta]$	84.50%	84.20%	84.50%	**84.60%**
$\mathbb{E}[r_T]$	16.0985%	**16.1744%**	16.1306%	16.1250%
VaR_α	3.1832%	3.2228%	3.2196%	**3.2237%**
CVaR_α	-0.3693%	-0.3731%	-0.3632%	**-0.3627%**
$\hat{\boldsymbol{\theta}}$	[0.11057, 0.34389, 0.17021, 0.13990, 0.22973, 0.00567]	[0.14408, 0.29763, 0.17649, 0.16160, 0.19700, 0.02320]	[0.12360, 0.29529, 0.19724, 0.15761, 0.2262, 0.00001]	**[0.12848, 0.29556, 0.18971, 0.15680, 0.22943, 0.00001]**

Table 1: Comparison of results obtained with the different approaches.

We have carried out other experiments by changing the random seed in the path simulation process. The results concerning KRR, SLSQP, and KRR+SLSQP are qualitatively the same.

Technical Details

Graphical Abstract



Cython

- Cython is a Python library that allows developers to write Python code that is translated and compiled in C extensions $\rightarrow$ C like performance.

Cython

- Cython is a Python library that allows developers to write Python code that is translated and compiled in C extensions $\rightarrow$ C like performance.
- The main improvement comes from the variables static type definitions.

Cython

- Cython is a Python library that allows developers to write Python code that is translated and compiled in C extensions $\rightarrow$ C like performance.
- The main improvement comes from the variables static type definitions.

Usage

- Create a .pyx file
- Write the code as in Python
- Add static type definitions for each variable
- Compile the .pyx code into a C extension
- Import the extension in the python script

Cython

- Cython is a Python library that allows developers to write Python code that is translated and compiled in C extensions $\rightarrow$ C like performance.
- The main improvement comes from the variables static type definitions.

Usage

- Create a .pyx file
- Write the code as in Python
- Add static type definitions for each variable
- Compile the .pyx code into a C extension
- Import the extension in the python script

```
def simulate(self, cnp.ndarray kappa,
                cnp.ndarray p, cnp.ndarray sigma, double T,
                cnp.ndarray N_list, cnp.ndarray event_type_list,
                cnp.ndarray event_direction_list,
                cnp.ndarray v_list, int batch_size):

        cdef double sum_kappa
        cdef cnp.ndarray pi, mu_vec, event_type, event_direction
        cdef double mu
        cdef int k, j, N, c, random_number
        cdef cnp.ndarray Rx, Ry, v
        cdef list pools, Rx_t, Ry_t, v_t, event_type_t,
                event_direction_t
        ...
```

```
HOW COMPILE THE CYTHON CODE:
python code/setup.py build_ext --inplace
```

Cython

- Cython is a Python library that allows developers to write Python code that is translated and compiled in C extensions $\rightarrow$ C like performance.
- The main improvement comes from the variables static type definitions.

Usage

- Create a .pyx file
- Write the code as in Python
- Add static type definitions for each variable
- Compile the .pyx code into a C extension
- Import the extension in the python script

```
def simulate(self, cnp.ndarray kappa,
                cnp.ndarray p, cnp.ndarray sigma, double T,
                cnp.ndarray N_list, cnp.ndarray event_type_list,
                cnp.ndarray event_direction_list,
                cnp.ndarray v_list, int batch_size):

        cdef double sum_kappa
        cdef cnp.ndarray pi, mu_vec, event_type, event_direction
        cdef double mu
        cdef int k, j, N, c, random_number
        cdef cnp.ndarray Rx, Ry, v
        cdef list pools, Rx_t, Ry_t, v_t, event_type_t,
                event_direction_t
        ...
```

```
HOW COMPILE THE CYTHON CODE:
python code/setup.py build_ext --inplace
```

	Cython	Standard
Time (sec)	207	260

Table 2: Execution time in Google Colab

Sampling procedure

- During the optimization procedure we must simulate price paths multiple times.

```
def simulate(self, kappa, p, sigma, T=1, batch_size=256):
    (...)
    for k in tqdm(range(batch_size)):
      N = np.random.poisson(lam = sum_kappa*T)
      (...)
      for j in range(N):
          event_type[j] = np.random.choice(len(kappa), p=pi)
          event_direction[j] = int(np.random.rand()<p[event_type[j]])
          if event_direction[j] == 0:
              mu = np.zeros(len(pools[k].Rx))
          else:
              mu = np.log(pools[k].Ry/pools[k].Rx)
          if event_type[j] == 0:
              v[j,:] = np.exp((mu-0.5*sigma[0]**2) + sigma[0]*np.random.randn()
                       )
          else:
              v[j,:] = np.zeros(len(pools[k].Rx))
              mu = mu[event_type[j]-1]
              v[j,event_type[j]-1] = np.exp((mu-0.5*sigma[event_type[j]]**2) +
                      sigma[event_type[j]]*np.random.randn())
          if event_direction[j] == 0:
              pools[k].swap_x_to_y(v[j,:])
          else:
              pools[k].swap_y_to_x(v[j,:]
      (...)
  return pools, Rx_t, Ry_t, v_t, event_type_t, event_direction_t
```

Sampling procedure

- During the optimization procedure we must simulate price paths multiple times.
- The random numbers must be always the same.

```
def simulate(self, kappa, p, sigma, T=1, batch_size=256):
    (...)
    for k in tqdm(range(batch_size)):
      N = np.random.poisson(lam = sum_kappa*T)
      (...)
      for j in range(N):
          event_type[j] = np.random.choice(len(kappa), p=pi)
          event_direction[j] = int(np.random.rand()<p[event_type[j]])
          if event_direction[j] == 0:
              mu = np.zeros(len(pools[k].Rx))
          else:
              mu = np.log(pools[k].Ry/pools[k].Rx)
          if event_type[j] == 0:
              v[j,:] = np.exp((mu-0.5*sigma[0]**2) + sigma[0]*np.random.randn()
                       )
          else:
              v[j,:] = np.zeros(len(pools[k].Rx))
              mu = mu[event_type[j]-1]
              v[j,event_type[j]-1] = np.exp((mu-0.5*sigma[event_type[j]]**2) +
                      sigma[event_type[j]]*np.random.randn())
          if event_direction[j] == 0:
              pools[k].swap_x_to_y(v[j,:])
          else:
              pools[k].swap_y_to_x(v[j,:]
      (...)
  return pools, Rx_t, Ry_t, v_t, event_type_t, event_direction_t
```

Sampling procedure

- During the optimization procedure we must simulate price paths multiple times.
- The random numbers must be always the same.
- We sampled them just once .

```
def simulate(self, kappa, p, sigma, T=1, batch_size=256):
    (...)
    for k in tqdm(range(batch_size)):
      N = np.random.poisson(lam = sum_kappa*T)
      (...)
      for j in range(N):
          event_type[j] = np.random.choice(len(kappa), p=pi)
          event_direction[j] = int(np.random.rand()<p[event_type[j]])
          if event_direction[j] == 0:
              mu = np.zeros(len(pools[k].Rx))
          else:
              mu = np.log(pools[k].Ry/pools[k].Rx)
          if event_type[j] == 0:
              v[j,:] = np.exp((mu-0.5*sigma[0]**2) + sigma[0]*np.random.randn()
                       )
          else:
              v[j,:] = np.zeros(len(pools[k].Rx))
              mu = mu[event_type[j]-1]
              v[j,event_type[j]-1] = np.exp((mu-0.5*sigma[event_type[j]]**2) +
                      sigma[event_type[j]]*np.random.randn())
          if event_direction[j] == 0:
              pools[k].swap_x_to_y(v[j,:])
          else:
              pools[k].swap_y_to_x(v[j,:]
      (...)
  return pools, Rx_t, Ry_t, v_t, event_type_t, event_direction_t
```

Sampling procedure

- During the optimization procedure we must simulate price paths multiple times.
- The random numbers must be always the same.
- We sampled them just once .
- We employed the python code for the sampling.

```
def simulate(self, kappa, p, sigma, T=1, batch_size=256):
    (...)
    for k in tqdm(range(batch_size)):
      N = np.random.poisson(lam = sum_kappa*T)
      (...)
      for j in range(N):
          event_type[j] = np.random.choice(len(kappa), p=pi)
          event_direction[j] = int(np.random.rand()<p[event_type[j]])
          if event_direction[j] == 0:
              mu = np.zeros(len(pools[k].Rx))
          else:
              mu = np.log(pools[k].Ry/pools[k].Rx)
          if event_type[j] == 0:
              v[j,:] = np.exp((mu-0.5*sigma[0]**2) + sigma[0]*np.random.randn()
                       )
          else:
              v[j,:] = np.zeros(len(pools[k].Rx))
              mu = mu[event_type[j]-1]
              v[j,event_type[j]-1] = np.exp((mu-0.5*sigma[event_type[j]]**2) +
                      sigma[event_type[j]]*np.random.randn())
          if event_direction[j] == 0:
              pools[k].swap_x_to_y(v[j,:])
          else:
              pools[k].swap_y_to_x(v[j,:]
      (...)
  return pools, Rx_t, Ry_t, v_t, event_type_t, event_direction_t
```

Sampling procedure

- During the optimization procedure we must simulate price paths multiple times.
- The random numbers must be always the same.
- We sampled them just once .
- We employed the python code for the sampling.

Samplings	One	Each iteration
Time (sec)	260	526

Table 3: Execution time in Google Colab

```
def simulate(self, kappa, p, sigma, T=1, batch_size=256):
    (...)
    for k in tqdm(range(batch_size)):
      N = np.random.poisson(lam = sum_kappa*T)
      (...)
      for j in range(N):
          event_type[j] = np.random.choice(len(kappa), p=pi)
          event_direction[j] = int(np.random.rand()<p[event_type[j]])
          if event_direction[j] == 0:
              mu = np.zeros(len(pools[k].Rx))
          else:
              mu = np.log(pools[k].Ry/pools[k].Rx)
          if event_type[j] == 0:
              v[j,:] = np.exp((mu-0.5*sigma[0]**2) + sigma[0]*np.random.randn()
                       )
          else:
              v[j,:] = np.zeros(len(pools[k].Rx))
              mu = mu[event_type[j]-1]
              v[j,event_type[j]-1] = np.exp((mu-0.5*sigma[event_type[j]]**2) +
                      sigma[event_type[j]]*np.random.randn())
          if event_direction[j] == 0:
              pools[k].swap_x_to_y(v[j,:])
          else:
              pools[k].swap_y_to_x(v[j,:]
      (...)
  return pools, Rx_t, Ry_t, v_t, event_type_t, event_direction_t
```

Conclusion

Summary & Conclusion

We were able to:

- Formulate the minimization problem as a two-step strategy.
- Exploit the efficiency of KRR and the flexibility of SLSQP.
- Reduce the computational load of paths simulation by sampling random numbers just once.
- Leverage the benefits of Cython to reduce the overall computational load.

Summary & Conclusion

We were able to:

- Formulate the minimization problem as a two-step strategy.
- Exploit the efficiency of KRR and the flexibility of SLSQP.
- Reduce the computational load of paths simulation by sampling random numbers just once.
- Leverage the benefits of Cython to reduce the overall computational load.

Further research

The main idea is to test the proposed approach for the CVaR minimization with other underlying simulation processes. This can be done in two ways:

- **Parametric approach**: Change the Poisson distribution of the events occurrence (e.g. Hawks process)
- **Non parametric approach**: Simulation of future scenarios using a generative neural network (e.g. GAN, Variational Autoencoder ecc.)

In the latter case, as we expect a huge drop in the simulation complexity, the KRR component could be substituted by a neural network to enhance the flexibility of the proposed methodology.

Thanks for your attention!