Getting Started With PyOptQuest

OptQuest is a simulation optimization engine built on a truly unique set of powerful algorithms and sophisticated analysis techniques including metaheuristics optimization, evolutionary algorithms, tabu search and scatter search, to name a few. We provide two interfaces to the OptQuest Engine which is written in Java. 1) The **`pyoptquest`** module provides wrappers around the Java classes so the full functionality of OptQuest can be accessed as if it were being called from a Java application. 2) The **`OptQuestOptimizer`** Python class is a wrapper around the OptQuest engine JAR, providing a Python interface to the engine.

Note that an OptQuest license is necessary in order to use the OptQuestEngine. Inquire at our website to get a trial license if you need one: https://www.opttek.com/products/optquest/

Installing PyOptQuest

The easiest way to install PyOptQuest is to use the **`pip`** package manager

```
> pip install pyoptquest
```

This will install all of the dependencies (except Java), including pyjnius which is used to communicate with Java. You will need to install your own copy of Java version 17. We use a Java Native Interface (JNI) bridge to the Java Virtual Machine (JVM) to map Python functions to the Java library. The hardest part of setting up PyOptQuest is getting your Python environment to connect with your installation of Java. Make sure that you have the Java directories on the path and that your Python environment can reach Java.

In the examples directory, we provide a test Python script that tests the Python-Java interface without any dependencies on OptQuest. If this works, you can be sure that your Python environment can see you Java installation

```
examples> python hello.py
Hello World!
```

Java-like Interface

With this interface, you will program in Python following the exact same patterns and exact same class names as you would in Java. That is, you can follow the OptQuest java documentation as written:

https://www.opttek.com/doc/v811engine/OptQuest_Engine_Documentation/OptQuest.htm.

This interface is perfect for the OptQuest expert who want the same Java patterns available in Python and needs full control over all of the optimization parameters.

Examples

In the examples directory, we provide a few Java-like examples written in Python. Since this interface maps 1:1 with the Java interface, it's best to use the detailed documentation from our website linked above.

Example Name	Description

Example Name	Description
`example_simple.py`	A very simple example that shows how to set up an optimization in Python and call the Java optimization.
`example_solver.py`	An example uses a string expression that's evaluated in Java rather than a Python function for the objective.
`example_frontier.py`	An example of a two-objective optimization.
`example_parallel.py`	An example of running multiple evaluators in parallel. With the Java-like interface, you need to manage the messaging between threads for parallel execution. The Pythonic interface is able to manage that for you, and is a much simpler way to get parallel evaluations.
`example_replication.py`	An example of a stochastic optimization where OptQuest will manage replications of the same solution.
`example_replicaton_with_confidence.py`	An advanced example of stochastic optimization where OptQuest checks the confidence interval of the solution to dynamically control the replications.

Pythonic Interface

The pythonic interface is designed to seamlessly plug into Simon Blanke's Gradient Free Optimizers, but does not require the Gradient Free Optimizer code to work.

Minimal Example

Here is an example using **`OptQuestOptimizer`** to minimize x^2 over the search space $x \in [-10, 10]$.

```
from pyoptquest import OptQuestOptimizer

# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'expression': 'x * x'}
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=10)

# print results
```

```
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
best score: 0.0
best parameters: {'x': 0.0}
Optimization time: 0.045327186584472656
```

The first step in designing an optimization is to define the **`search_space`**. The **`search_space`** is a dictionary of variable names mapped to their properties. In the example, we simply define a continuous variable that can take on values between -10 and 10. You can view all supported variable types and their properties in the Supported Variable Types section.

At least one objective must be defined and given an optimization **`type`**. An objective can be of the type **`min`** (minimize) or **`max`** (maximize). The **`'expression'`** property indicates that the objective will be represented by a simple mathematical expression provided by the user as a string. In this case, the expression is **`x * x`** or x^2.

Once a search space and objective have been defined it is possible to run a minimal optimization. Note that the license ID and OptQuest JAR file path must be provided when creating the optimization. Finally, call **`search()`** with a specified number of iterations (**`n_iter`**) on the optimizer object (**`opt`**) and the optimization will begin.

The output shows that **`OptQuestOptimizer`** found the best solution. The optimal value and solution can be obtained from the optimizer object after the optimization by calling **`best_score()`** and **`best_para()`**.

Example with Simulation Evaluator

`OptQuestOptimizer` is tailored toward optimizing simulated problems rather than purely mathematical ones.

When designing an optimization, we are not limited to specifying simple mathematical objectives; we can provide a custom evaluator callback function that uses inputs given by **`OptQuestOptimizer`** and returns output values. The output values can then be used by objectives. For example, the custom evaluator could call an external process to start a simulation with the provided input values. The sole purpose of output values is to have a means of collecting information from the evaluator. Inputs are passed to the evaluator, and outputs are collected from it. Note that this is a distinct type of evaluator from constraint and objective evaluators, which return a boolean value or single numerical value, respectively.

Here is an example using an evaluator function named **`simulation`**; a small modification on the previous example:

```
from pyoptquest import OptQuestOptimizer


def simulation(inputs):
    # get inputs from OptQuestOptimizer
    x = inputs['x']

    # run the "simulation"
    y = x ** 2
```

```
    # create dict to hold return values from the simulation
    outputs = {'y': y}
    return outputs


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# specify the output(s) of the simulation evaluator (the output(s) of the simulation() function)
output_space = ['y']

# define the objective(s)
objectives = {
    'obj': {'type': 'min', 'expression': 'y'}  # minimize the output "y"
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    evaluator=simulation,
    output_space=output_space,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=10)

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
best score: 0.0
best parameters: {'x': 0.0, 'y': 0.0}
Optimization time: 0.0624852180480957
```

The result is the same as the previous examples, but we used a user-supplied evaluator callback function to generate an output **y** which was then used by the objective **obj**.

Example with Objective Evaluator

Objectives are not limited to being string expressions and can be defined with a function evaluator that takes **inputs** and optionally **outputs** as parameters. The example below shows an objective evaluator callback function that only takes **inputs** as a parameter:

```
from pyoptquest import OptQuestOptimizer
```

```
# function to evaluate the objective
def objective_evaluator(inputs):
    x = inputs['x']
    return x ** 2


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'evaluator': objective_evaluator}  # specify the evaluator
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=10)

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

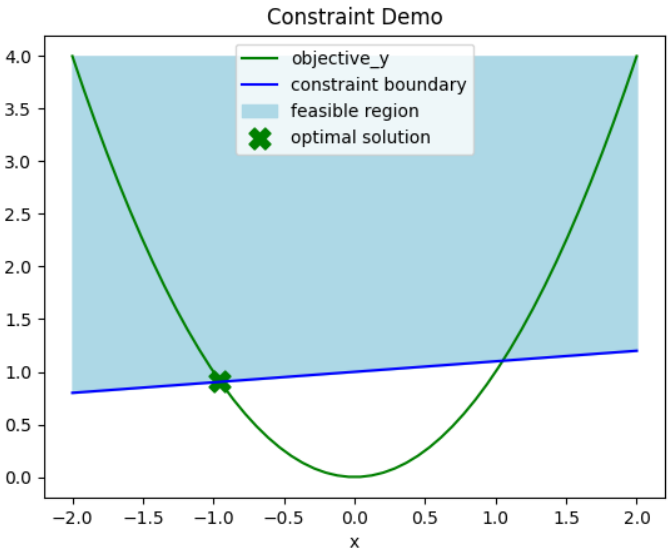
```
best score: 0.0
best parameters: {'x': 0.0}
Optimization time: 0.09617185592651367
```

The result is the same as the previous examples, but we used a user-supplied objective evaluator function instead of a simple mathematical string expression.

The difference between an objective evaluator callback function and an an evaluator callback function is that the objective evaluator can only return a single value that can't be used by other objectives while an evaluator can return multiple outputs that can be used by multiple objectives.

Constraints Example

Consider the Minimal Example where we are trying to minimize the objective **y**=x^2 for $x \in [-10, 10]$. x is constrained to be on the interval $[-10, 10]$, but we can specify more complex constraints. Let's add the constraint that the objective **y**= x^2 must be above the line $0.1x + 1$. The problem now looks like this:



Here is how this problem can be solved using **`OptQuestOptimizer`**:

```
from pyoptquest import OptQuestOptimizer

# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'expression': 'x * x'}
}

constraints = {
    'my_constraint': {'expression': 'x * x ≥ 0.1 * x + 1'}
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    constraints=constraints,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=100)

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
best score: 0.9164935413654939
best parameters: {'x': -0.9573366917472107}
Optimization time: 0.13147902488708496
```

We can also see all the solutions that were evaluated:

```
# print all solutions
print('all solutions:')
print(opt.search_data)
```

Output:

```
all solutions:
    iteration  replication  feasible           y          x
0           1            1     False    0.000000   0.000000
1           2            1      True  100.000000 -10.000000
2           3            1      True  100.000000  10.000000
3           4            1      True   25.000000  -5.000000
4           5            1      True   25.000000   5.000000
..        ...          ...       ...         ...        ...
95         96            1      True   20.200825  -4.494533
96         97            1      True   49.104083  -7.007431
97         98            1      True   50.107366   7.078656
98         99            1      True   89.333619   9.451646
99        100            1     False    0.013505   0.116211

[100 rows x 5 columns]
```

Notice the third column which indicates whether a solution was in the feasible region or not.

Multiple constraints can be defined in **`constraints`**, but they must all have unique names. A constraint can also be the result of a callback function that returns a boolean instead of a string expression. Here is an example of a problem with an evaluated constraint:

```
from pyoptquest import OptQuestOptimizer


def domain_constraint(inputs):
    x = inputs['x']
    return abs(x) ≥ 1


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}
```

```
# define the objective(s)
objectives = {
    'y': {'type': 'min', 'expression': 'x * x'}
}

constraints = {
    'domain': {'evaluator': domain_constraint},
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    constraints=constraints,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=300)

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
best score: 1.0
best parameters: {'x': 1.0}
Optimization time: 0.2471938133239746
```

The optimal solutions are at $x = 1$ and $x = -1$.

Parallel Evaluation Example

Parallelization is easy when using the optimizer. Simply pass the number of parallel executions you'd like to run to **`OptQuestOptimizer.search()`**, specify the number of **`parallel_evaluators`** i.e. processes you'd like to have running, and that's it.

If we want to parallelize the Minimal Example we can simply change the line:

```
opt.search(n_iter=10)  # non-parallel execution
```

to:

```
opt.search(n_iter=100, parallel_evaluators=2)  # parallel execution
```

Remember, while Python supports threads and parallel execution semantics, Python code itself does not run concurrently. So parallel evaluation will only be a time-saver if you spawn external processes, like long-running simulations.

Here is a made up example of a problem that benefits from parallelization:

```
import time
from pyoptquest import OptQuestOptimizer


# represents a large simulation running for one second
def sleep_evaluator(inputs):
    time.sleep(1)
    return {}


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'expression': 'pow(x - pi, 2)'}
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    evaluator=sleep_evaluator,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=100, parallel_evaluators=20)  # increasing parallel_evaluators will decrease the
optimization time

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
best score: 6.007550195017685e-07
best parameters: {'x': 3.1408175697107796}
Optimization time: 5.143232583999634
```

Since the **`sleep_evaluator`** function executes for one second and we do 100 iterations, 20 at a time, the simulation takes about five seconds to execute (1 second * 100 iterations / 20 in parallel.)

Replications Example

Replications are extra evaluations executed on the same input(s) for the purpose of accounting for variability in simulations. It is recommended to use parallel evaluators when running replications on large simulations.

`OptQuestOptimizer` has support for replications of an evaluation for a given set of inputs, but it does not provide any sort of random seed.

The user can set a fixed number of replications to be executed, or a variable number of executions. To set a variable number of replications, a minimum and maximum number of replications is specified. **`OptQuestOptimizer`** will execute the minimum number of replications and then continue running replications until each objective has reached 95% confidence or the maximum number of replications have been executed.

To set a fixed number of replications, simply call **`OptQuestOptimizer.search()`** like this:

```
opt.search(n_iter=10, replications=20)  # run 20 replications
```

To set a variable number of replications, pass a tuple to **`OptQuestOptimizer.search()`** containing the minimum and maximum number of replications, respectively:

```
opt.search(n_iter=10, replications=(5, 20))  # run between 5 and 20 replications
```

Here is an example of an optimization with variable replications (note the use of a status monitor to track the progress):

```
import random

from pyoptquest import OptQuestOptimizer


# optional function for tracking the status of the optimization, called every iteration/replication
def status_monitor(inputs, outputs, objectives, iteration, replication):
    print(f'--executing {iteration}, replication {replication}')


# return a random number close to 1 so that a random number of replications are called for by
OptQuest
def objective_evaluator(dec_var_values, output_values):
    return 1 + (random.random() - 0.5) * 0.2


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'evaluator': objective_evaluator}
}
```

```
# define the optimization
opt = OptQuestOptimizer(search_space, objectives,
                        status_monitor=status_monitor,
                        license_id=999999999,
                        optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=2, replications=(5, 20))
# opt.search(n_iter=10, replications=20)  # fixed replications

print()

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
--executing 1, replication 1
--executing 1, replication 2
--executing 1, replication 3
--executing 1, replication 4
--executing 1, replication 5
--executing 1, replication 6
--executing 1, replication 7
--executing 1, replication 8
--executing 1, replication 9
--executing 2, replication 1
--executing 2, replication 2
--executing 2, replication 3
--executing 2, replication 4
--executing 2, replication 5
--executing 2, replication 6
--executing 2, replication 7

best score: 1.0059779289574402
best parameters: {'x': 0.0}
Optimization time: 0.06905317306518555
```

Notice that for the first iteration of inputs, confidence was met after nine replications. For the second iteration of inputs it only took seven replications before confidence was met.

Advanced Variable Replications

When using variable replications OptQuest will continue to execute replications until it executes the max replications or until confidence is met for all objectives. By default, confidence is met when, over all executed replications, the objective value is within 5% of the mean 95% of the time.

However, some objectives don't require the same amount confidence that others do. You have the ability to change the confidence parameters for each objective when defining them.

Below is an example of an objective defined with user-specified confidence parameters:

```
import random

from pyoptquest import OptQuestOptimizer


# optional function for tracking the status of the optimization, called every iteration/replication
def status_monitor(inputs, outputs, objectives, iteration, replication):
    print(f'--executing {iteration}, replication {replication}')


# return a random number close to 1 so that a random number of replications are called for by
OptQuest
def objective_evaluator(dec_var_values, output_values):
    return 1 + (random.random() - 0.5) * 0.2


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'evaluator': objective_evaluator, 'confidence': 5, 'error': 0.04}
    # confidence 5 means 99% confidence interval
}

# define the optimization
opt = OptQuestOptimizer(search_space, objectives,
                        status_monitor=status_monitor,
                        license_id=999999999,
                        optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=2, replications=(5, 20))

print()

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
--executing 1, replication 1
--executing 1, replication 2
--executing 1, replication 3
--executing 1, replication 4
--executing 1, replication 5
--executing 1, replication 6
--executing 1, replication 7
--executing 1, replication 8
--executing 1, replication 9
--executing 1, replication 10
```

```
--executing 2, replication 1
--executing 2, replication 2
--executing 2, replication 3
--executing 2, replication 4
--executing 2, replication 5
--executing 2, replication 6
--executing 2, replication 7
--executing 2, replication 8
--executing 2, replication 9
--executing 2, replication 10
--executing 2, replication 11
--executing 2, replication 12
--executing 2, replication 13
--executing 2, replication 14
--executing 2, replication 15
--executing 2, replication 16

best score: 0.9853533444387222
best parameters: {'x': -10.0}
Optimization time: 0.04686570167541504
```

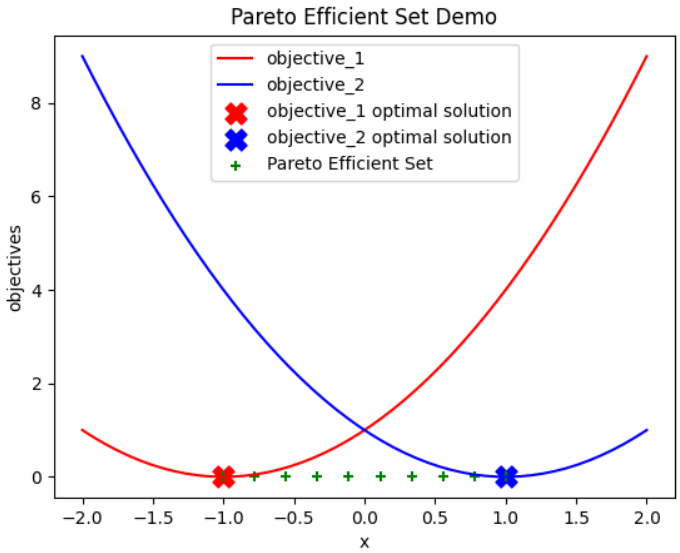
We set **`confidence`** to **`5`** and **`error`** to **`0.04`**; we tightened the confidence required on this objective. Confidence 5 means we want a 99% confidence interval, and an error of 0.04 means that the grouping around the mean has a tolerance of 4%. For this objective confidence will be met when the objective value is within 4% of the mean 99% of the time.

In the previous example only 9 and 7 replications were run before confidence was met. In this example 10 and 16 replications were run before confidence was met because there was tighter tolerance on the objective and more replications were needed in order to reach that tolerance.

Multi-Objective Example

`OptQuestOptimizer` can optimize for multiple objectives. For example, consider a problem where you're solving for x and you want to minimize **`objective_1`**$= (x+1)^2$ and also minimize **`objective_2`**$= (x-1)^2$. The minima are $x=-1$ and $x=1$, respectively. However, there is no single best solution x for both objectives. The idea of an optimal solution is replaced by a Pareto efficient set; a set of optimal solutions which are better than all other solutions, but not necessarily better than each other.

Consider the following image of the problem:



The two minima for the two objectives are marked with X's. Points in the Pareto efficient set are marked with green crosses. Note that the Pareto efficient set is infinite here and the marked points are just a finite sample of it.

These Pareto efficient solutions represent the best solutions; notice that any point less than (left of) **`objective_1`**'s minimum at $x = -1$ is worse than all Pareto efficient solutions for both objectives, and any point greater than (right of) **`objective_2`**'s minimum at $x = 1$ is also worse than all Pareto efficient solutions for both objectives. Points in the Pareto efficient set are not necessarily optimal for either individual objective, but they are a compromise between the two objectives.

Here is an example demonstrating how to solve this problem using **`OptQuestOptimizer`**:

```
from matplotlib import pyplot as plt

from pyoptquest import OptQuestOptimizer

# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -2, 'max': 2}
}

# define the objective(s)
objectives = {
    'objective_1': {'type': 'min', 'expression': 'pow(x + 1, 2)'},
    'objective_2': {'type': 'min', 'expression': 'pow(x - 1, 2)'}
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')

# do the optimization
opt.search(n_iter=10)
```

```
# print results
print('Pareto front:')
print(opt.best_score)
print('Optimization time:', opt.optimization_time)
```

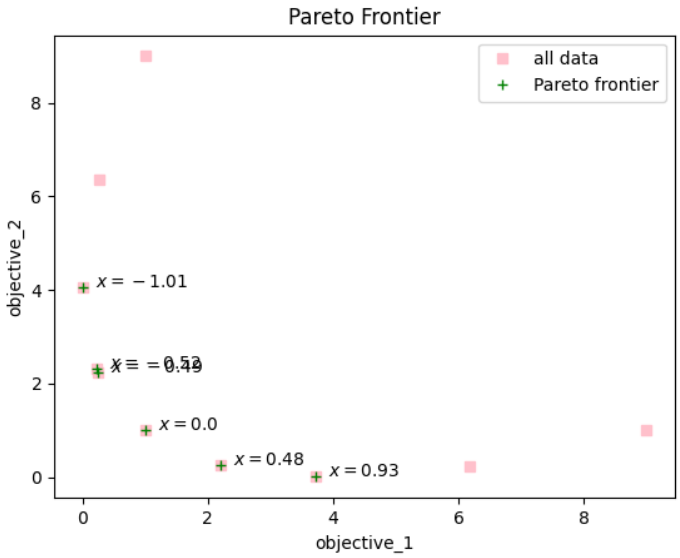
Output:

```
pareto frontier data:
   iteration  replication  objective_1  objective_2         x
0          1            1     1.000000     1.000000  0.000000
1          4            1     0.000160     4.050825 -1.012666
2          5            1     0.231362     2.307359 -0.518999
3          6            1     3.728985     0.004753  0.931058
4          8            1     0.255932     2.232344 -0.494103
5         10            1     2.198158     0.267683  0.482619
Optimization time: 0.05336761474609375
```

An analyst making a decision about a multi-objective problem would probably benefit from seeing the objectives against each other to aid in making a decision. When plotting objectives against eachother, the Pareto efficient set is called the Pareto frontier. Let's make this plot:

```
# plot results
fig, ax = plt.subplots()
ax.set_title('objective_1 vs objective_2')
ax.set_xlabel('objective_1')
ax.set_ylabel('objective_2')
opt.search_data.plot(x='objective_1', y='objective_2', style='s', color='pink', ax=ax)  # all data
opt.best_score.plot(x='objective_1', y='objective_2', style='+', color='green', ax=ax)  # Pareto
frontier
ax.legend(['all data', 'pareto frontier'])
# label points on the Pareto front
for idx, (x_coord, y_coord) in enumerate(zip(opt.best_score['objective_1'],
opt.best_score['objective_2'])):
    x = round(opt.best_score["x"][idx], 2)
    ax.annotate(text=str(f'$x={x}$'), xy=(x_coord + 0.2, y_coord))
plt.show()
```

Output:



This plot show us the tradeoffs between Pareto frontier solutions to our problem (the green crosses.) The solution at $x = 0.0$ looks like a good compromise for both objectives. Picking a different solution such as $x = 0.48$ minimizes **objective_2** but increases **objective_1** even more so, and we're trying to minimize both objectives. However, depending on what **objective_1** and **objective_2** represent, we may favor $x = 0.48$. This is where the human factor comes in; with multiple objectives we cannot mathematically say that one solution on a Pareto frontier is better than another, the Pareto frontier is a tool representing optimal objective tradeoffs that a human analyst can use to help make a decision.

Status Monitor

The **status_monitor** callback function is passed to **OptQuestOptimizer** and gets called after every time an iteration or replication has been evaluated. It's useful for tracking the status of the optimization while it's running.

Here is an example using **status_monitor** in the Minimal Example:

```
from pyoptquest import OptQuestOptimizer


# optional function for tracking the status of the optimization (called every iteration)
def status_monitor(inputs, outputs, objectives, iteration, replication):
    print(f'--status monitor says hello from iteration {iteration} with objective value
{objectives["y"]}')


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'expression': 'x * x'}
}

# define the optimization
opt = OptQuestOptimizer(
```

```
    search_space,
    objectives,
    status_monitor=status_monitor,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=10)

print()

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
--status monitor says hello from iteration 1 with objective value 0.0
--status monitor says hello from iteration 2 with objective value 100.0
--status monitor says hello from iteration 3 with objective value 100.0
--status monitor says hello from iteration 4 with objective value 25.63732245730393
--status monitor says hello from iteration 5 with objective value 21.671725343768035
--status monitor says hello from iteration 6 with objective value 55.06539502557916
--status monitor says hello from iteration 7 with objective value 6.103445395617211
--status monitor says hello from iteration 8 with objective value 57.78743065485578
--status monitor says hello from iteration 9 with objective value 5.823021408526737
--status monitor says hello from iteration 10 with objective value 36.80563848228416

best score: 0.0
best parameters: {'x': 0.0}
Optimization time: 0.0624849796295166
```

The **status_monitor** function must be able to receive the parameters **inputs**, **outputs**, **objectives**, **iteration**, and **replication**; even if the it isn't using these parameters.

Prematurely Stopping an Optimization

OptQuestOptimizer accepts a **user_stop** callback function that returns a boolean which tells it to continue or stop the optimization every iteration/replication. The function takes the same parameters as the **status_monitor** callback. Below is an example of a **user_stop** function being used to stop the optimization at 5 iterations:

```
from pyoptquest import OptQuestOptimizer


# optional function for tracking the status of the optimization (called every iteration)
def status_monitor(inputs, outputs, objectives, iteration, replication):
    print(f'--status monitor says hello from iteration {iteration} with objective value
{objectives["y"]}')


# return True if the optimization should be stopped
```

```
def user_stop(inputs, outputs, objectives, iteration, replication):
    if iteration ≥ 5:
        print('-- STOPPING! --')
        return True
    else:
        return False


# define the input(s)
search_space = {
    'x': {'type': 'continuous', 'min': -10, 'max': 10}
}

# define the objective(s)
objectives = {
    'y': {'type': 'min', 'expression': 'x * x'}
}

# define the optimization
opt = OptQuestOptimizer(
    search_space,
    objectives,
    status_monitor=status_monitor,
    user_stop=user_stop,
    license_id=999999999,
    optquest_jar=r'../OptQuest.jar')
opt.search(n_iter=10)

print()

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
--status monitor says hello from iteration 1 with objective value 0.0
--status monitor says hello from iteration 2 with objective value 100.0
--status monitor says hello from iteration 3 with objective value 100.0
--status monitor says hello from iteration 4 with objective value 25.63732245730393
--status monitor says hello from iteration 5 with objective value 21.671725343768035
-- STOPPING! --

best score: 0.0
best parameters: {'x': 0.0}
Optimization time: 0.03778338432312012
```

The **`user_stop`** callback function has access to the current solution being run and can also easily be configured to stop the optimization after a certain amount of time has passed or after some other event occurs.

Gradient Free Optimizers Support

OptQuestOptimizer is compatible with Gradient Free Optimizers (GFO). For example, the optimizer in the convex_function.py demo can easily be swapped to **OptQuestOptimizer**.

HillClimbingOptimizer example:

```
import numpy as np
from gradient_free_optimizers import HillClimbingOptimizer


def convex_function(pos_new):
    score = -(pos_new["x1"] * pos_new["x1"] + pos_new["x2"] * pos_new["x2"])
    return score


search_space = {
    "x1": np.arange(-100, 101, 0.1),
    "x2": np.arange(-100, 101, 0.1),
}

opt = HillClimbingOptimizer(search_space)  # using HillClimbingOptimizer
opt.search(convex_function, n_iter=300000)
```

Output:

```
Results: 'convex_function'
   Best score: -6.462348535570529e-23
   Best parameter:
      'x1' : -5.6843418860808015e-12
      'x2' : -5.6843418860808015e-12

   Evaluation time   : 9.109798669815063 sec    [33.31 %]
   Optimization time : 18.237623929977417 sec    [66.69 %]
   Iteration time    : 27.34742259979248 sec    [10969.96 iter/sec]
```

OptQuestOptimizer example:

```
import numpy as np
from pyoptquest import OptQuestOptimizer


def convex_function(pos_new):
    score = -(pos_new["x1"] * pos_new["x1"] + pos_new["x2"] * pos_new["x2"])
    return score


search_space = {
```

```
    "x1": np.arange(-100, 101, 0.1),
    "x2": np.arange(-100, 101, 0.1),
}

opt = OptQuestOptimizer(  # using OptQuestOptimizer
    search_space,
    license_id=999999999,
    optquest_jar=r'optquest.jar')
opt.search(convex_function, n_iter=300000)

# print results
print('best score:', opt.best_score)
print('best parameters:', opt.best_para)
print('Optimization time:', opt.optimization_time)
```

Output:

```
best score: -6.462348535570529e-23
best parameters: {'x1': -5.6843418860808015e-12, 'x2': -5.6843418860808015e-12}
Optimization time: 0.3541069030761719
```

In the above example note that, like GFO, **`OptQuestOptimizer`** doesn't need to be defined with **`objectives`** if **`OptQuestOptimizer.search()`** is passed an objective evaluator e.g. **`convex_function`**. If an objective is passed like this, **`OptQuestOptimizer`** will maximize the objective.

`OptQuestOptimizer` Constructor Parameters

Below is a description of each parameter that can be passed to the **`OptQuestOptimizer`** constructor.

- **`search_space`**: Required parameter. The inputs/decision variables for the optimization. This should be a **`dict`** of variable names mapped to dictionaries of their corresponding properties.
- **`objectives`**: Not required for the constructor. If no objective is supplied here than an objective evaluator must be passed to **`OptQuestOptimizer.search()`** later on. This should be a **`dict`** of objective names mapped to dictoinaries of their corresponding properties.
- **`evaluator`**: Not required. A callback function which should take an **`input`** parameter; a **`dict`** of variable names and values. The function should then run the user logic e.g. perform a simulation, and then return an **`output`** **`dict`** of output names mapped to their values. The names (keys) in the **`output`** return **`dict`** must be those specified by **`output_space`**, described next.
- **`output_space`**: Sometimes required. Only used if the constructor's **`evaluator`** parameter is passed an evaluator callback function since it will generate outputs. This should be a **`list`** containing the names of all outputs that will be collected from the evaluator callback function.
- **`constraints`**: Not required. Provides a way to put constraints on the inputs and outputs of the problem. This should be a **`dict`** mapping constraint names to dictionaries of their corresponding properties. See the constraints example for more details.
- **`status_monitor`** and **`user_stop`**: callback functions that must take the parameters **`inputs`**, **`outputs`**, **`objectives`**, **`iteration`**, and **`replication`**. See the corresonding sections for more details on these callback functions: Status Monitor and Prematurely Stopping an Optimization.
- **`license_id`**: The ID number for your OptQuest license. The trial license is used in the provided examples. An optimization running with the trial license is limited to 7 variables and 500 iterations. The trial license ID number is

`999999999`.

- `OptQuestJar`: A path to an `OptQuest.jar` file of version 9.1.1.2 or higher.

`OptQuestOptimizer.search()` Parameters

Below is a list of parameters that can be passed to the `OptQuestOptimizer.search()` function.

- `objective_evaluator`: If no `objectives` parameter is passed to the `OptQuestOptimizer` constructor then this parameter must be passed to the `OptQuestOptimizer.search()` function as a callback function. The callback function must take an `inputs` parameter which expects a `dict` mapping input variable names to their values and it can optionally take a second parameter `outputs` which maps output names to their values.
- `n_iter`: The number of iterations of unique input combinations to try.
- `replications`: The number of times to replicate an iteration of unique inputs. Replications are only useful when the user's `evaluator` is stochastic/non-deterministic and inputs need to be evaluated multiple times. The value for this parameter can be an `int` which specifies the number of replications that should be done for each iteration or the value can be a `tuple` which specifies the minimum and maximum number of replications that will be done for each iteration e.g. `(5, 20)`. See Replications Example for more details.
- `parallel_evaluators`: The maximum allowed number of concurrent calls to the `evaluator` callback function (which was passed to the `OptQuestOptimizer` constructor.) The `evaluator` function must be reentrant and if it uses shared state, it is responsible for managing the synchronization (e.g., `mutex`es) as needed.
- `suggested_runs`: A dict containing solutions to evaluate as the first iterations. See `oqo_example_suggested_runs.py` for a demonstration.

Supported Variable Types

Continuous

A continuous variables is simply a real value, e.g. *3.14159*.

Properties

- `min`: the lower bound (inclusive) on the values the variable can take on.
- `max`: the upper bound (inclusive) on the values the variable can take on.

Discrete

Discrete variables are continuous variables that are constrained to values at step intervals, e.g. *1.5, 3, 4.5, 6* for a step size of *1.5*.

Properties

- `min`: Lower bound (inclusive) on the values the variable can take on. The variable can necessarily take on this value as well.
- `max`: Upper bound on the values the variable can take on. The variable can take on this value only if `max` - `min` is an integer multiple of `step`.
- `step`: The step size; the range of values the variable can take is `min`, `min` + 1*`step`, `min` + 2*`step`, `min` + 3*`step`, etc.

Integer

An integer variable can only take real integer values.

Properties

- *min*: the lower bound (inclusive) on the values the variable can take on.
- *max*: the upper bound (inclusive) on the values the variable can take on.

Binary

Can only take on values 0 and 1.

Enumeration

An enumeration variable is limited to taking on values from an enumerated list of values e.g. *{7, 10, 25}*.

Properties

- *values*: the enumerated list of values the variable can take on. The format can be an iterable with numeric members, a comma delimited string of values, or a space delimited string of values.

Permutation

When you define a permutation type in **`search_space`** a permutation group is created. A permutation group contains multiple permutation variables. Each variable will be an integer value representing that variable's index within the group (starting at index 1.)

Properties

- *elements*: The argument for this property can either be an integer or a list of strings. An integer will specify the number of elements (permutation variables) the permutation group should have. If a list of strings is passed, then each string in the list becomes the name of a permutation variable in the group.

Demonstration of Variable Types

Here's a cheat sheet demonstrating how each variable type can be defined:

```
search_space = {
    'continuous_var': {'type': 'continuous', 'min': -5, 'max': 5},
    'discrete_var': {'type': 'discrete', 'min': -1.5, 'max': 4.5, 'step': 1.5},
    'integer_var': {'type': 'integer', 'min': -3, 'max': 3},
    'binary_var': {'type': 'binary'},
    'enumeration_var1': {'type': 'enumeration', 'values': [1, 2, 3]},
    'enumeration_var2': {'type': 'enumeration', 'values': np.array([1, 2, 3])},
    'enumeration_var3': {'type': 'enumeration', 'values': '1, 2, 3'},
    'permutation_grp_A': {'type': 'permutation', 'elements': ['one', 'two', 'three']},
    'permutation_grp_B': {'type': 'permutation', 'elements': 3}
}
```

Expression Syntax

Expressions can be used when defining objectives or constraints.

When defining a constraint, the expression must contain a comparison operator. The supported comparison operators are `=`, `≤`, and `≥`.

The following functions can be used when creating string expressions:

Function	Syntax	Description
min	`min(x,y)`	Returns the smaller of two numbers.
max	`max(x,y)`	Returns the larger of two numbers.
sqrt	`sqrt(x)`	Returns the square root of a number.
log	`log(x)`	Returns the natural logarithm of a specified number.
log10	`log10(x)`	Returns the base 10 logarithm of a specified number.
pow	`pow(x,y`	Returns a specified number raised to the specified power.
exp	`exp(x)`	Returns e raised to the specified power
abs	`abs(x)`	Returns the absolute value of a specified number.
rand	`rand()`	Returns a random number between 0 and 1, inclusive.
fmod	`fmod(x,y)`	Returns the remainder of x / y.
floor	`floor(x)`	Returns the largest whole number less than or equal to the specified number.
ceil	`ceil(x)`	Returns the smallest whole number greater than or equal to the specified number.
sin	`sin(x)`	Returns the sine of x, where x is an angle in radians.
cos	`cos(x)`	Returns the cosine of x, where x is an angle in radians.
tan	`tan(x)`	Returns the tangent of x, where x is an angle in radians.
sinh	`sinh(x)`	Returns the hyperbolic sine of x, where x is an angle in radians.
cosh	`cosh(x)`	Returns the hyperbolic cosine of x, where x is an angle in radians.
tanh	`tanh(x)`	Returns the hyperbolic tangent of x, an angle in radians.
asin	`asin(x)`	Returns the arcsine of x in the range –π/2 to π/2.
acos	`acos(x)`	Returns the arccosine of x in the range 0 to π.
atan	`atan(x)`	Returns the arctangent of x in the range of –π/2 to π/2 radians.
atan2	`atan2(x,y)`	Returns the arctangent of y/x in the range –π to π radians. If both parameters of atan2 are 0, the function returns 0.
DtoR	`DtoR(x)`	Converts degrees to radians.

Function	Syntax	Description
RtoD	`RtoD(x)`	Converts radians to degrees.

Additionally, the mathematical constants **pi** and **e** are valid expression syntax.

Examples

The following examples are provided in the examples directory and most are used in this documentation.

Example Name	Description
oqo_example_parabola.py	The simplest possible example of an optimization with **OptQuestOptimizer**.
oqo_example_simulation.py	Same as **parabola_example.py** but demonstrates how an external simulation can be used.
oqo_example_objective_evaluator.py	Demonstrates how to use an objective evaluator callback function.
oqo_example_constraint.py	Demonstrates how to define a mathematical constraint.
oqo_example_constraint_evaluator.py	Demonstrates how to define a constraint as a boolean callback function.
oqo_example_parallel_evaluations.py	Demonstrates how to enable parallel evaluators.
oqo_example_permutation.py	An example permutation problem optimizing the order of transitions of production machines.
oqo_example_replications.py	A simple demonstration using a fixed number of replications.
oqo_example_replications_with_confidence.py	An advanced demonstration using a variable number of replications and a custom objective confidence configuration.
oqo_example_frontier.py	A demonstration of a problem with multiple objectives.
oqo_example_status_monitor.py	Demonstration of **status_monitor** callback function usage for tracking the progress of an optimization.
oqo_example_user_stop.py	Demonstration of **user_stop** callback function usage for prematurely stopping an optimization that's in progress.
oqo_example_suggested_runs.py	Demonstration of the **suggested_runs** parameter in the **OptQuestOptimizer.search()** function.
oqo_example_all_variables.py	Shows how to define and use all the variable types supported by the OptQuestOptimizer
oqo_example_gfo_convex.py **oqo_example_gfo_non_convex.py**	Examples of passing an objective evaluator to **OptQuestOptimizer.search()** in GFO style.
oqo_example_cell_tower_location.py	A simple example with a rectangular solution space.
oqo_example_cell_tower_polygon_location.py	An advanced example with a non-trivial polygon solution space. Requires the **shapely** package.

Example Name	Description
`oqo_example_radio_antenna.py`	An advanced example with a non-trivial polygon solution space, multiple constraints, and usage of many **`OptQuestOptimizer`** features. Requires the **`shapely`** package. This example does not work with the demo license supplied in the example. You need to use your license.