OPTIONPAYOFFER

A tool for option trading

Team Member:
Chenxi Xiang 186003830
Peiwen Zhang 174009097
Tianyi Ji 190003115

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Content

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Application Description

Our application: OptionPayOffer, is an option portfolio payoff calculation tool with visualizations.

This tool will be able to:

- Estimate the different spot-based curves of vanilla portfolios.
- Provide option pricing based on Black-Scholes model with Monte-Carlo simulations under different level of spot price.
- Evaluate option price under different market using different evaluation engine.

Users will be able to:

- Add and edit option trades: call/put options, strike price, quantity of trade and premium.
- Choose visualizations for option: Payoff Curve, Net Payoff Curve, P&L Curve, PV Curve, Delta Curve and Gamma Curve
- Set pricing parameters, precision and methods, including annual risk-free rate, volatility, dividend yield, spot price, maturity, etc.
- Access to the help function, including the description for each input parameters, such as, Qty-unit of each instrument.

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Target User and Application Impact

Our application targets on (include but not limit to):

- Industry professionals who need a quick look, pricing, estimate and visualizations on options.
- College students who are learning and understanding options objectives.
- Investors who seek to invest in relevant products and comparing different product by themselves.

The application impact to the industry will be based on our target users:

- Industry users: This application can help them save time on writing formulas, changing engines for simulations and realizing visualizations, which will increase their working efficiency.
- College students: This application can help them to deliver a first understanding on option pricing and its dynamics by modifying option trades and accessing to several types of payoff curves. Also, this time-saving application can help them to confirm calculations and generate visualizations.
- Investors: This application can help the non-professional investors, investors new to this instrument and investors are interested in this instrument to get a before-hand knowledge before they come to their financial advisors, brokers or dealers, which increases information transparency and information equality.

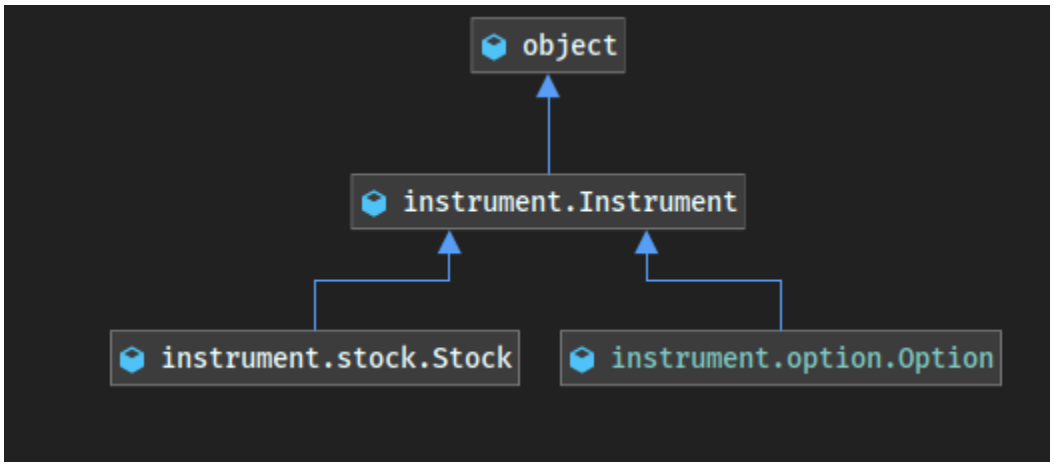
OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

System Design

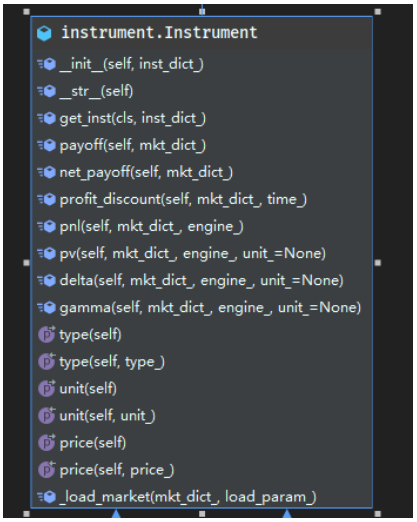
Data Structure

It is important for us to design a suitable data structure to hold our assets as well as portfolios. And It is also necessary to design an efficient data structure to provide a stable environment when we try to import some large data set.

For Assets:



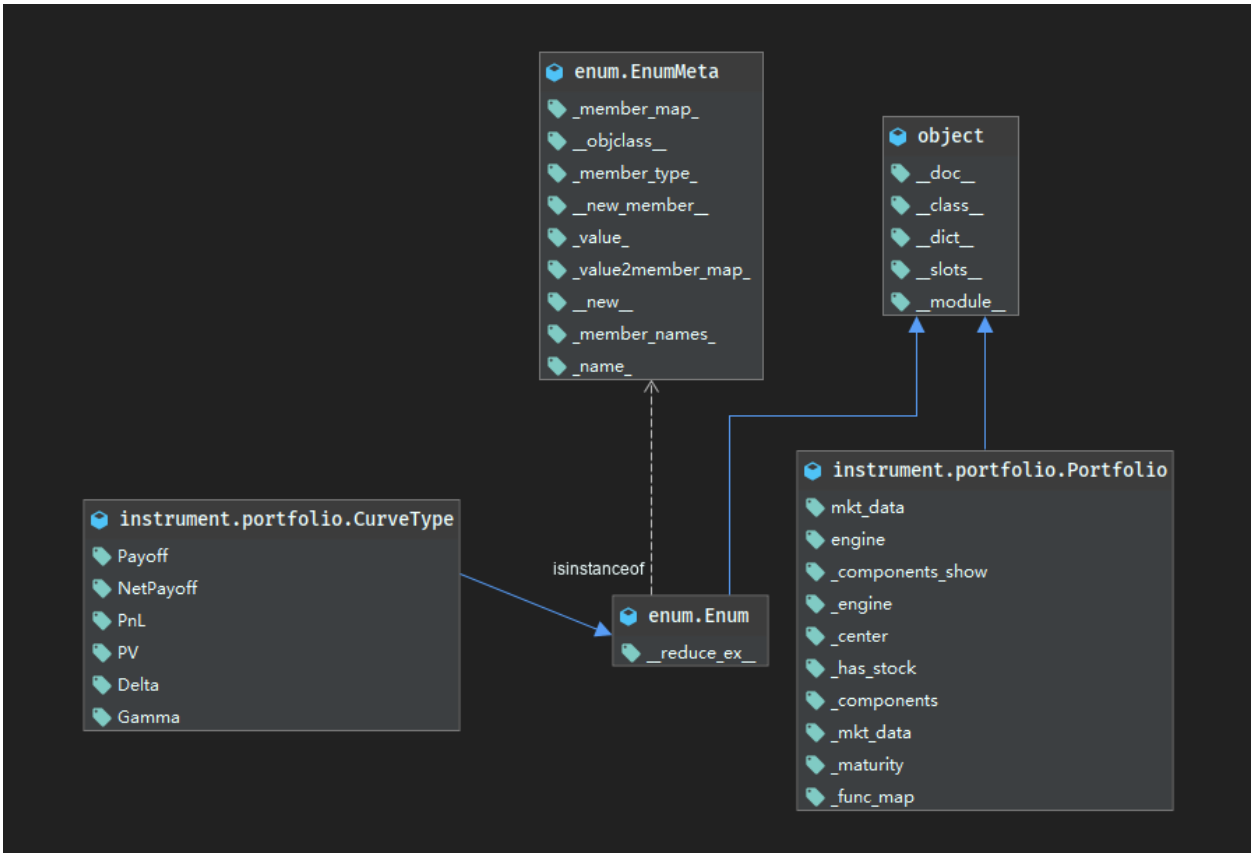
We have a base class named instrument to generate some common properties and methods for option class and stock class.



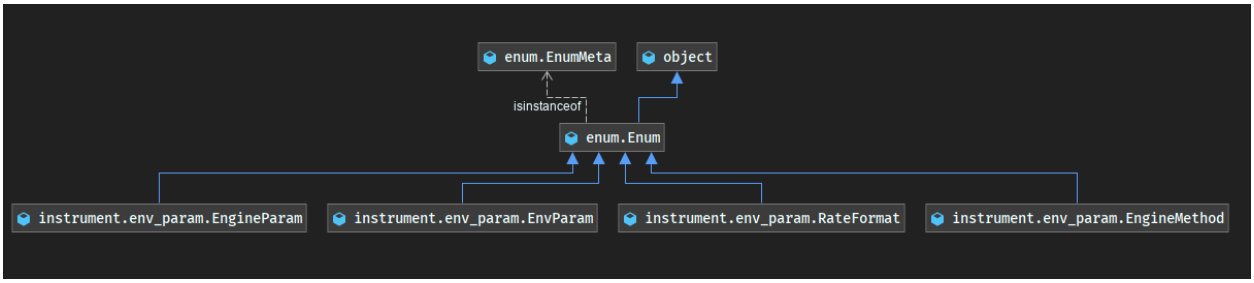
For Portfolios:

We have a class called portfolio to hold all options and stocks input from users. And it also have some functions to calculate some essential data.

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang



For Environment Settings:
We tend to let users set their preferences and parameters before they actually trading. We design a environment setting module independent from single option setting module. What's more, we provide a default parameter set by generating a instance of env_param.

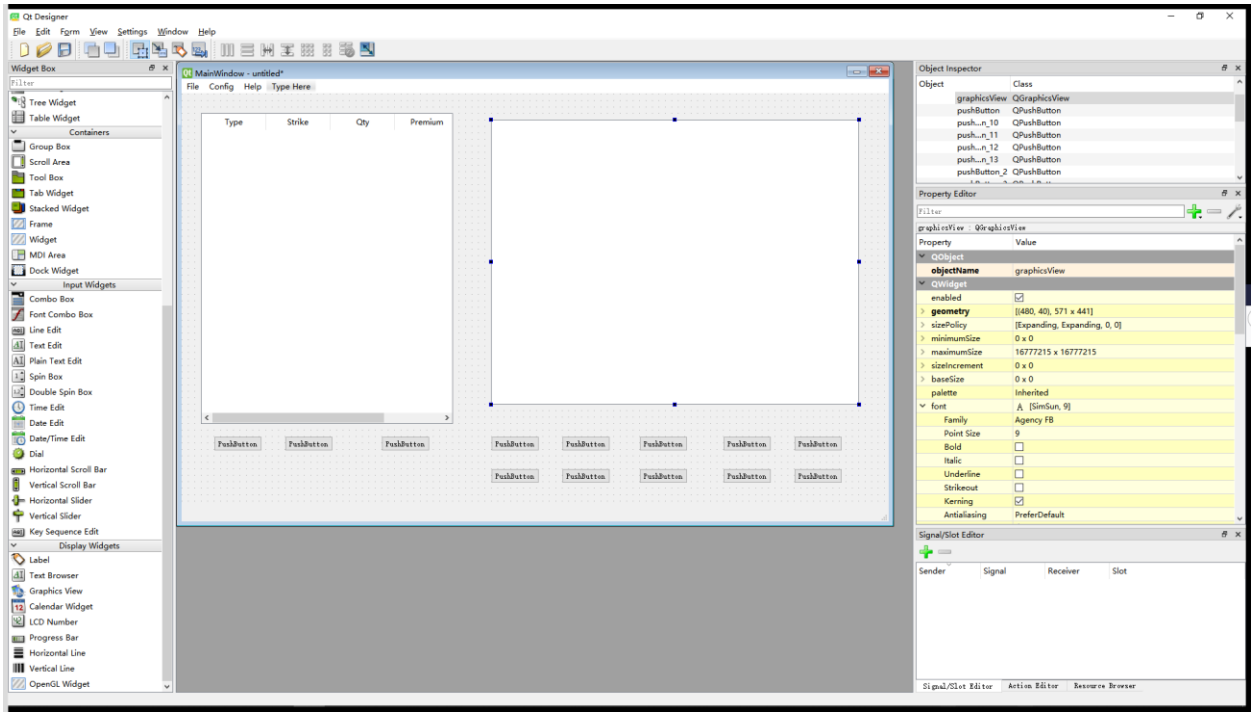


GUI Design

Our project bases on PyQt 5.12.0 with the help of QtDesigner, QtUIC and QtRCC.

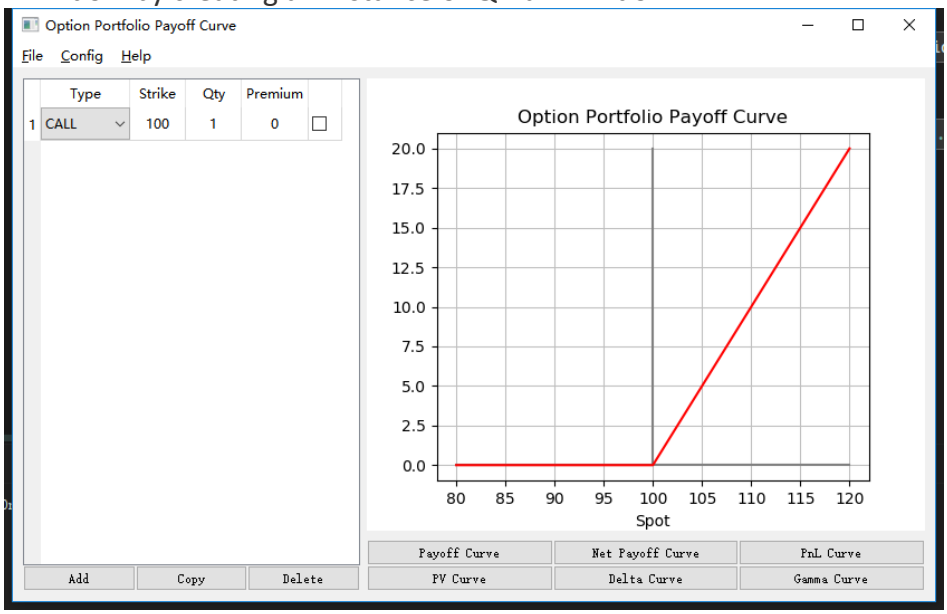
OptionPayOffer

Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

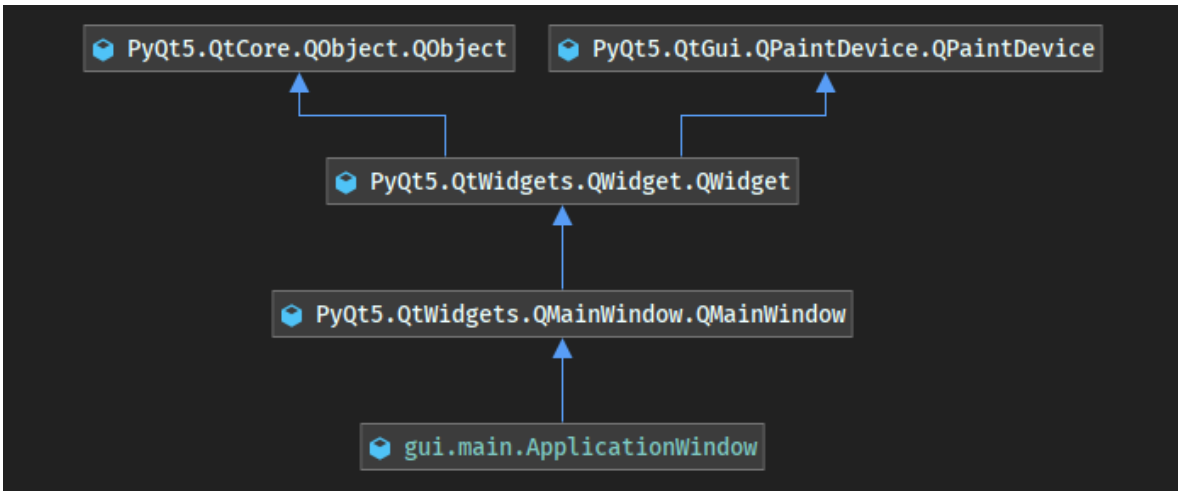


For MainWindow:

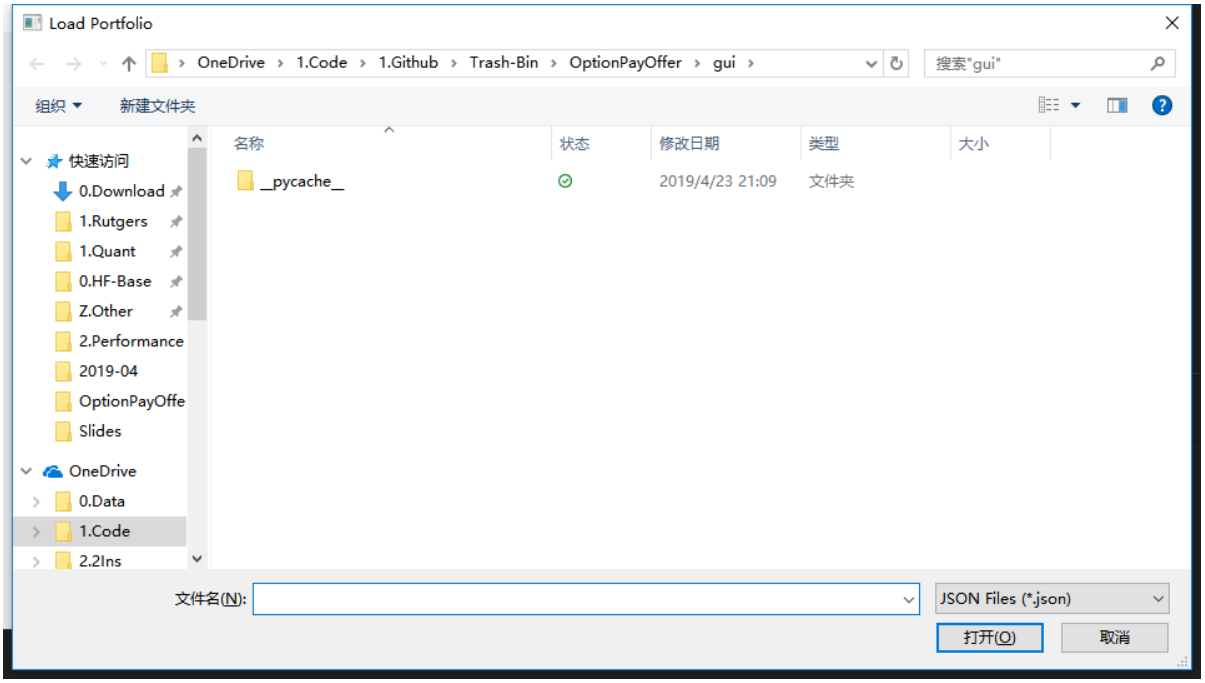
We get main window by creating an instance of QMainWindow.



OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

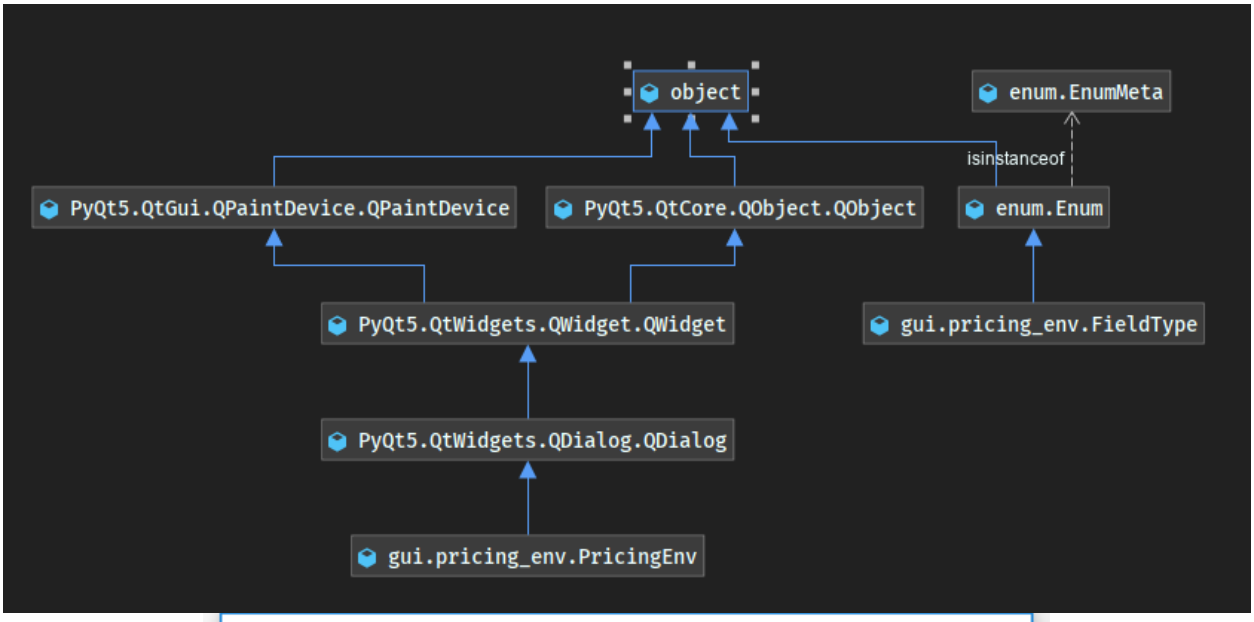


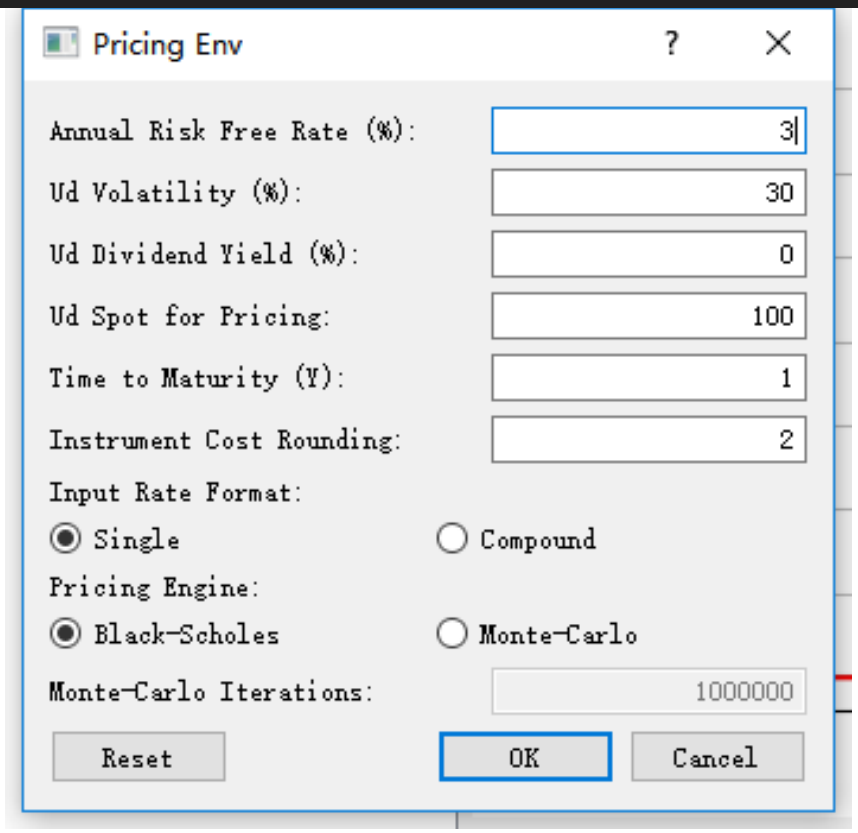
For Saving / Loading / Experting module:
It is quit easy to fulfill the requirement of this module. Just call PyQt5.QtWidgets.QFileDialog.getOpenFileName and it will automatically show the dialog we want.



For Setting Dialog:
There is a button we created for calling a sub-class of QDialog called PricingEnv. And the structure of PricingEnv shows below:

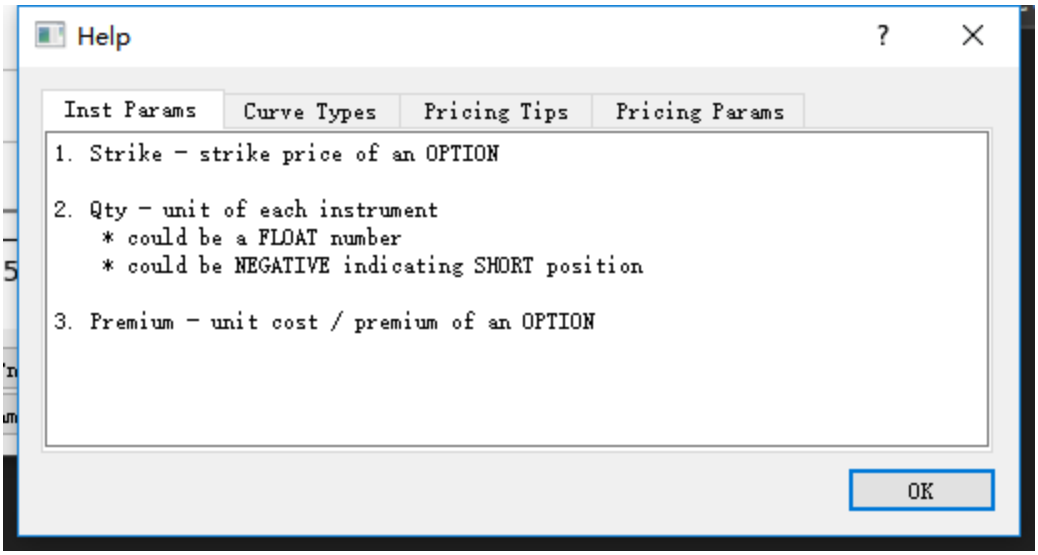
OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang



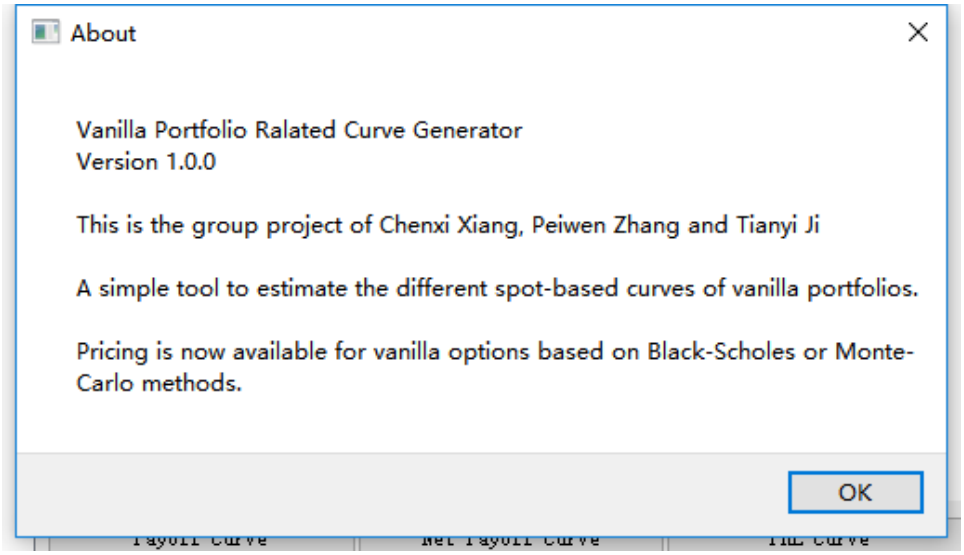


For more information:
There is a dialog to show how numbers we needed calculated and what the economical meaning for each greek and statistic.

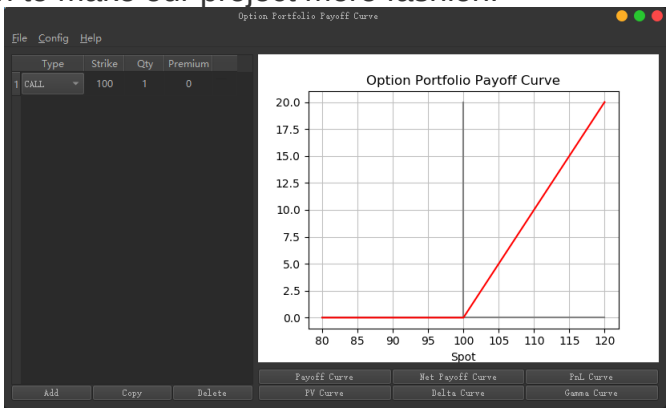
OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang



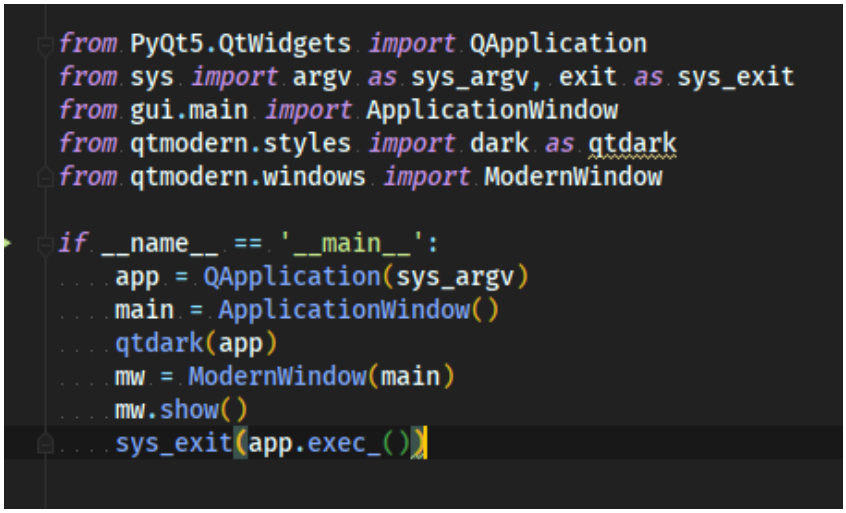
There is also an about dialog to show the description of our program and our teammates.



For Dialogs' Style:
We call the qtmodern to make our project more fashion.



OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang



```python
from PyQt5.QtWidgets import QApplication
from sys import argv as sys_argv, exit as sys_exit
from gui.main import ApplicationWindow
from qtmodern.styles import dark as qtdark
from qtmodern.windows import ModernWindow

if __name__ == '__main__':
    app = QApplication(sys_argv)
    main = ApplicationWindow()
    qtdark(app)
    mw = ModernWindow(main)
    mw.show()
    sys_exit(app.exec_())
```

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Performance Analysis

Computing Complexity

For Calculating each Greeks and numbers like PnL, the module follows their own computing complexity. Generally, Calculating Greeks takes O(n) for each option. And for the overall portfolio, calculating corresponding numbers takes O(N^2).

Running Time

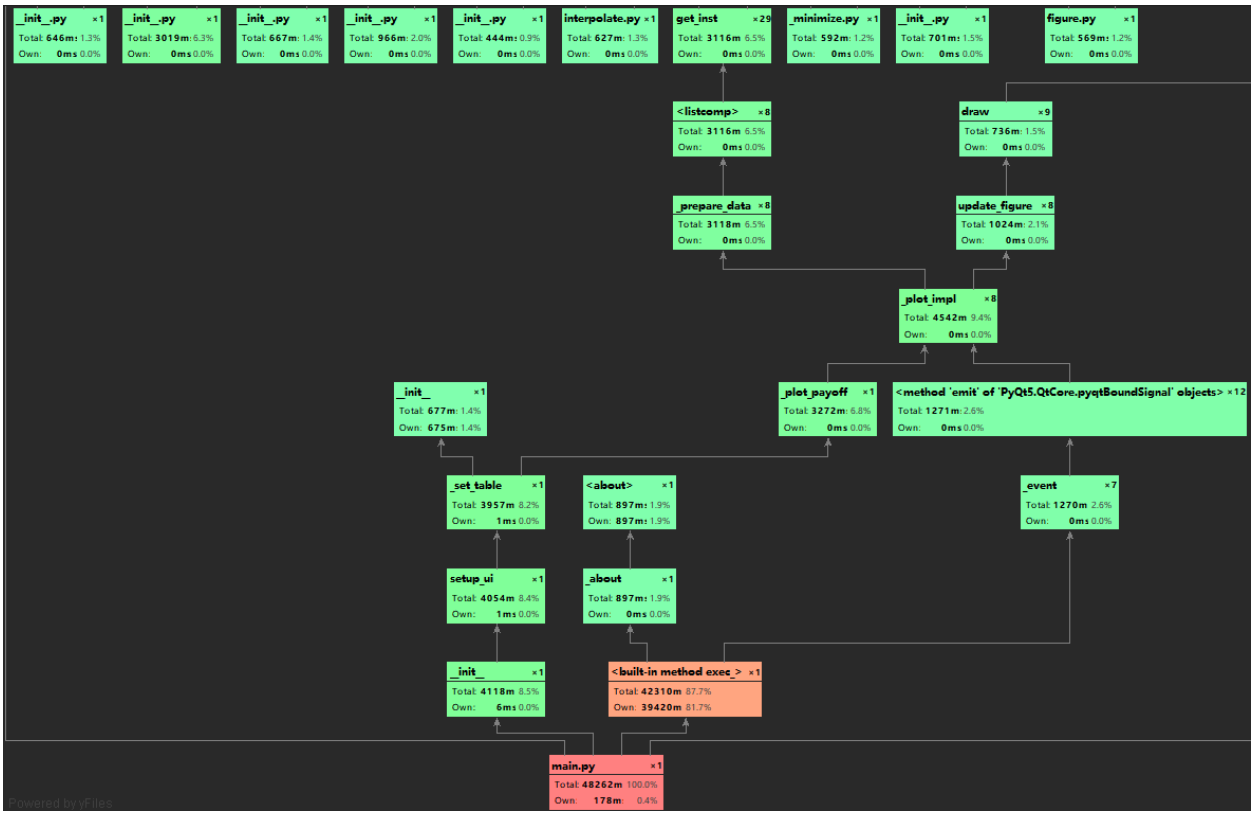
The profile statistics shows below (for the most time-consuming parts):

Name	Call Count	Time (ms)	Own Time (ms)
main.py	1	48262 100.0%	178 0.4%
<built-in method exec_>	1	42310 87.7%	39420 81.7%
_find_and_load_unlocked	402	4629 9.6%	3 0.0%
_find_and_load	402	4629 9.6%	6 0.0%
_load_unlocked	383	4622 9.6%	4 0.0%
_call_with_frames_removed	540	4589 9.5%	0 0.0%
_plot_impl	8	4542 9.4%	0 0.0%
exec_module	315	4358 9.0%	2 0.0%
__init__	1	4118 8.5%	6 0.0%
setup_ui	1	4054 8.4%	1 0.0%
_set_table	1	3957 8.2%	1 0.0%
_plot_payoff	1	3272 6.8%	0 0.0%
<built-in method builtins.__import__>	102	3247 6.7%	0 0.0%
_prepare_data	8	3118 6.5%	0 0.0%
<listcomp>	8	3116 6.5%	0 0.0%
get_inst	29	3116 6.5%	0 0.0%
_handle_fromlist	2110	3113 6.5%	3 0.0%

The Call Graph of OptionPayOffer shows below:

OptionPayOffer

Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang



Conclusion

For Summary

The option has been becoming an increasingly popular investment tool in either speculation and risk hedging for financial institutions and even individual investors. However, huge risks are embedded in options trading, and there is potential for costly mistakes. Our product, OptionPayOffer, helps investors mitigate the risk of making mistakes in option pricing, by providing users with access to payoff visualization, quick and accurate options pricing as well as Greeks calculation. Users have great flexibility to try out different combinations among options and stocks to meet their investment or risk-hedging objectives. Additionally, they can choose different engines to price different types of options. Therefore, OptionPayOffer will help our users make fast and safe option trading decisions without worrying about making mistakes in complicated mathematics.

For Future Improvement

There are three main points to improve our program in the future.
Firstly, we can add the support of real options data with connection to the MySQL database with data in it. It will show the real option Greeks curves to help traders analyze options portfolios better.

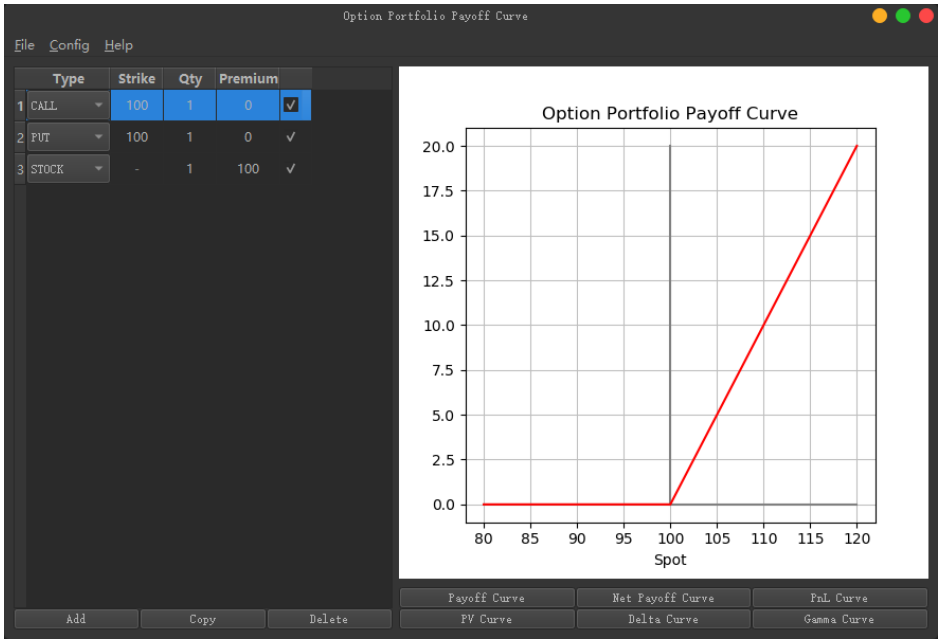
Secondly, the program will contain an interesting trading module for traders to improve their sense of arbitrage with virtual options.

Finally, our program will support more types of different options rather than vanilla options only.

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

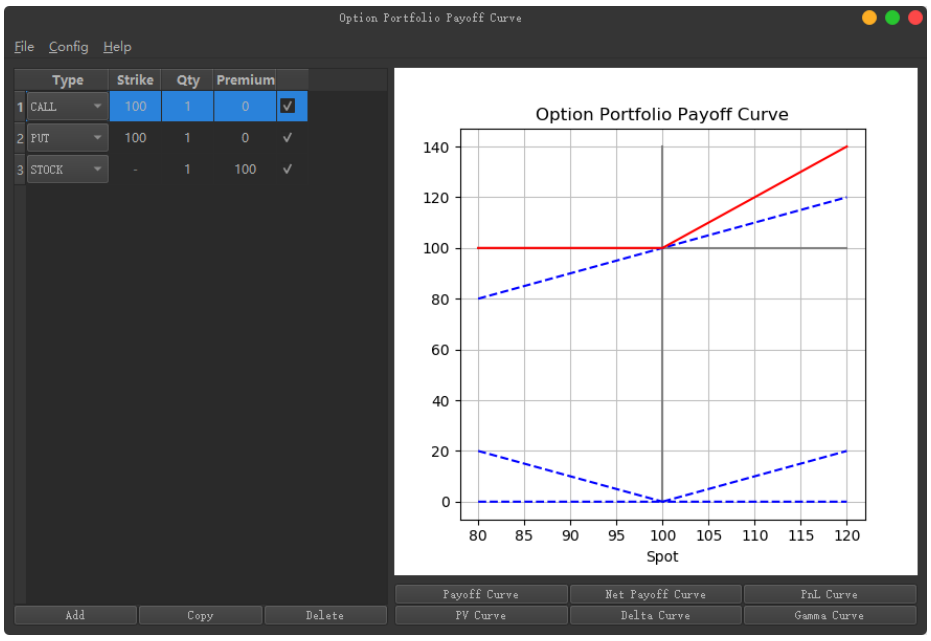
Output

A portfolio contains a call, put and stock:



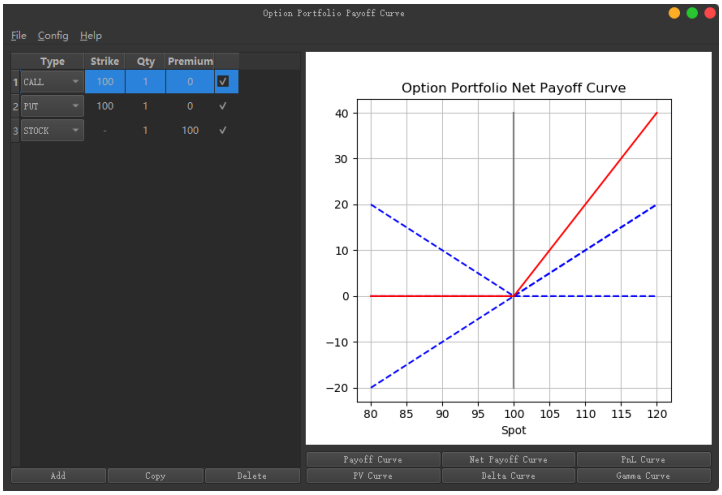
Plot some curves we want:

Payoff:

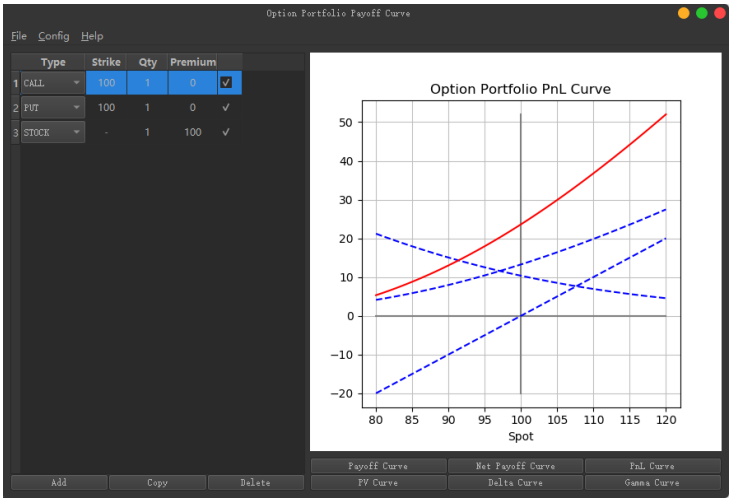


OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

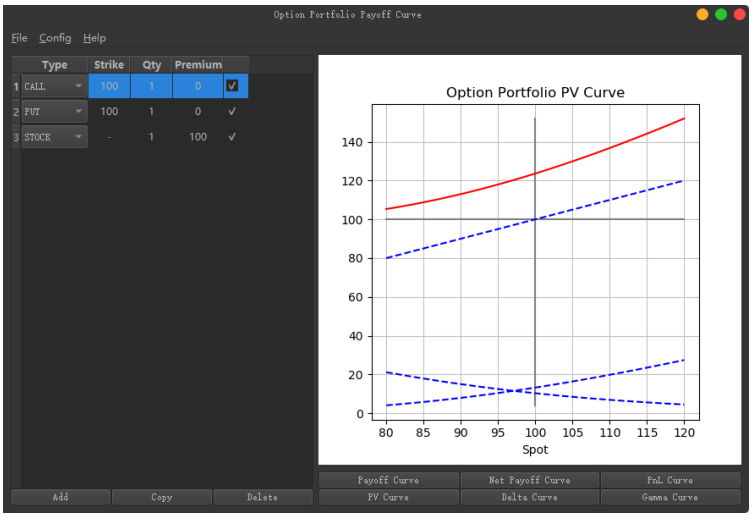
Net Payoff:



PnL Curve:



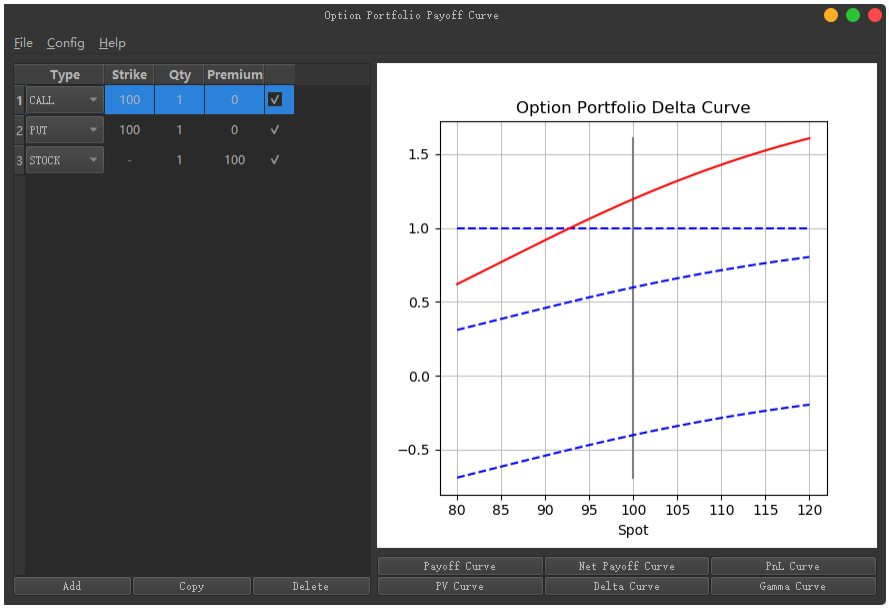
PV Curve:



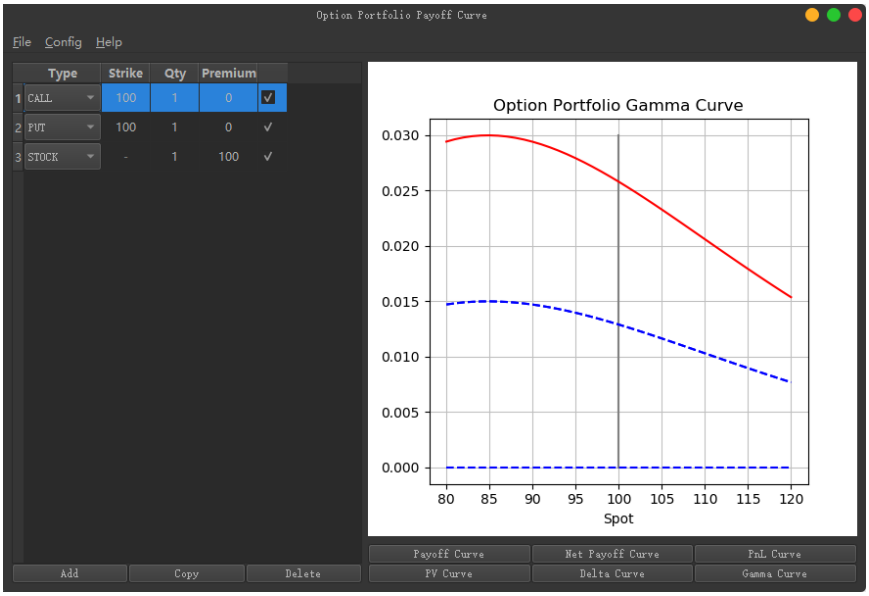
OptionPayOffer

Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Delta Curve:

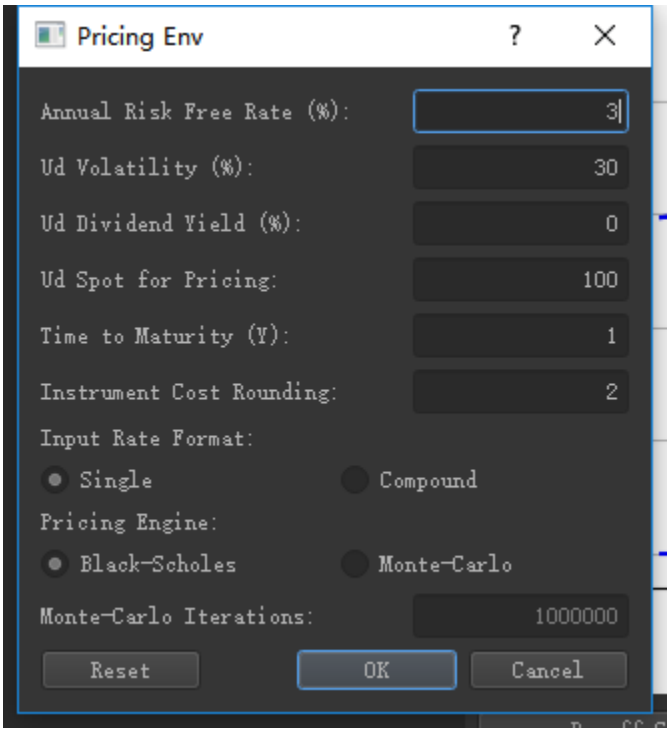


Gamma Curve:

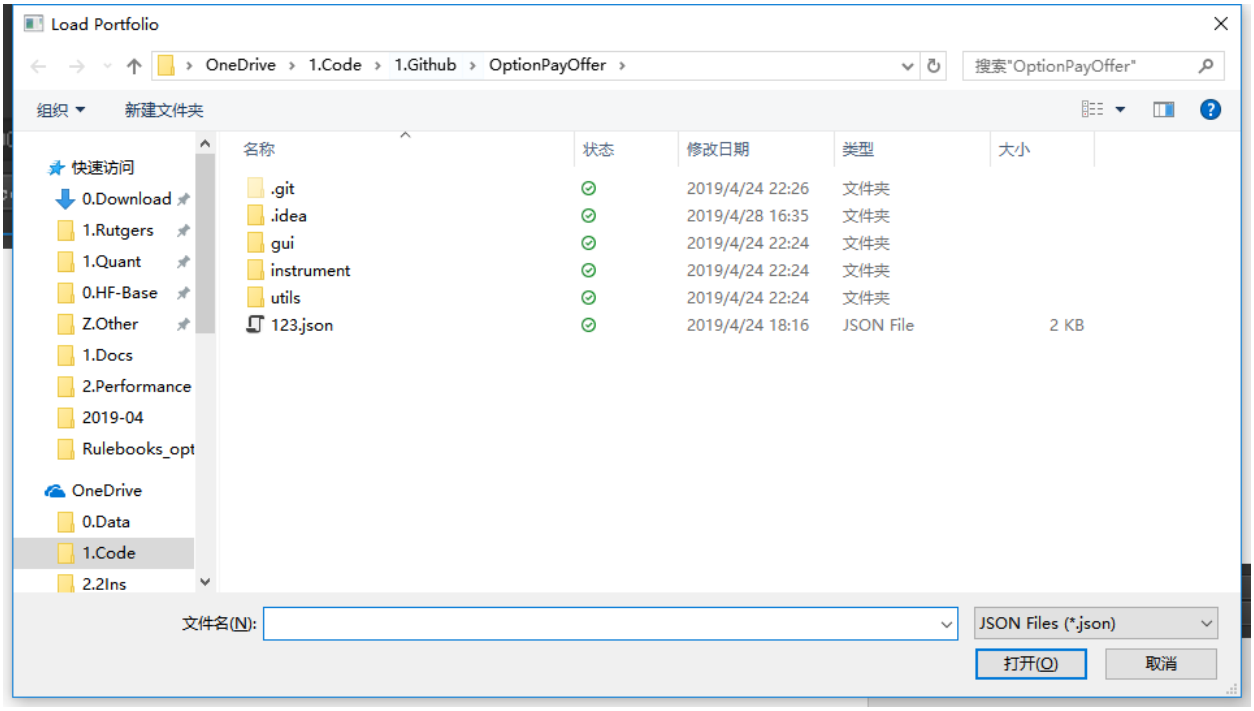


OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Set / Reset pricing environment

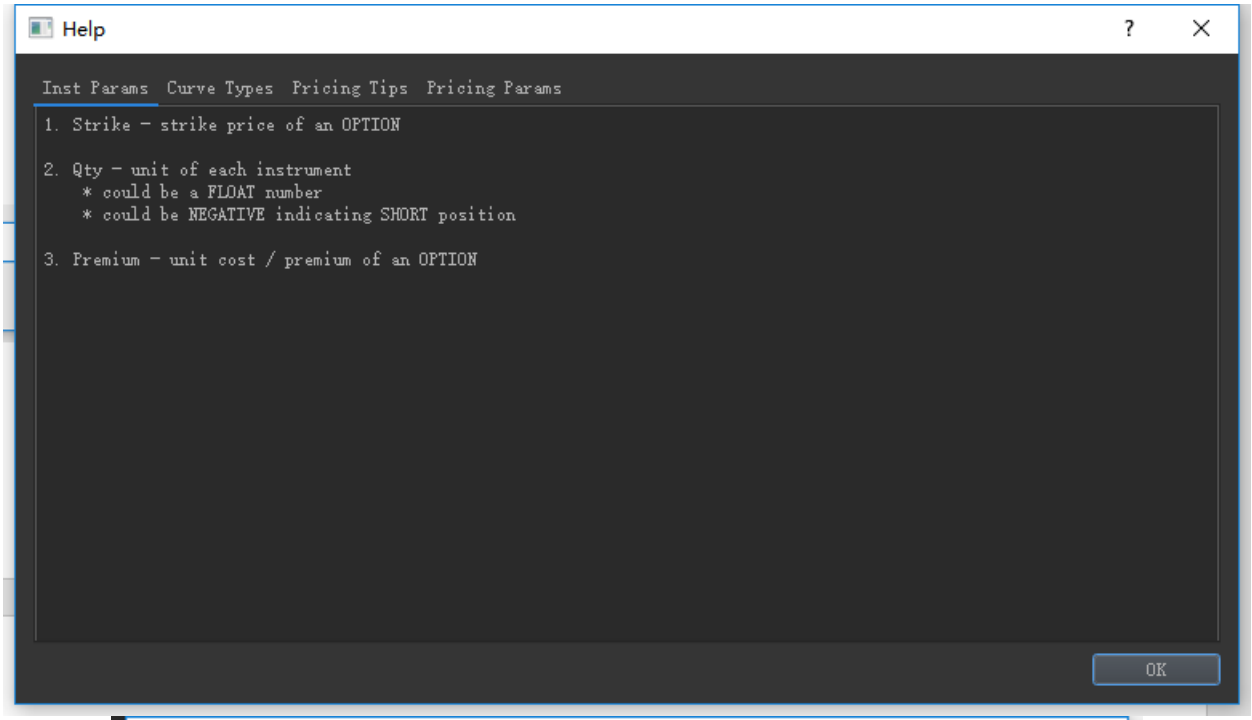


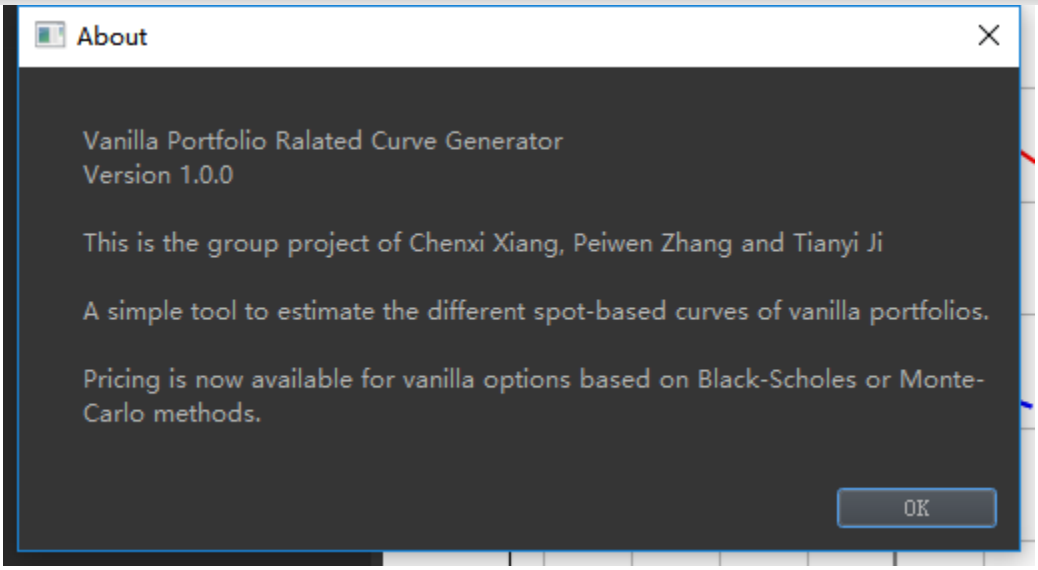
Load / Save / Expert Curves



OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Help / About





OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

Appendix

Source Code:

- Main.py

```
from PyQt5.QtWidgets import QApplication
from sys import argv as sys_argv, exit as sys_exit
from gui.main import ApplicationWindow
from qtmodern.styles import dark as qtdark
from qtmodern.windows import ModernWindow

if __name__ == '__main__':
    app = QApplication(sys_argv)
    main = ApplicationWindow()
    qtdark(app)
    mw = ModernWindow(main)
    mw.show()
    sys_exit(app.exec_())
```

- gui__init__.py is empty

- gui\custom.py

```
# coding=utf-8
"""customized widgets"""

from PyQt5.QtCore import Qt, pyqtSignal
from PyQt5.QtWidgets import QCheckBox, QComboBox, QPushButton, QRadioButton, QSizePolicy,
QTableWidget
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
# from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar
from matplotlib.figure import Figure


class CustomPushButton(QPushButton):
    """customized push button to return widget name and value when current check state is changed"""
    pressed = pyqtSignal(str)

    def __init__(self, display_='CustomPushButton', signal_='', *args, **kwargs):
        super(CustomPushButton, self).__init__(display_, *args, **kwargs)
        self._signal = signal_
        self.clicked.connect(self._event)

    def _event(self):
        self.pressed.emit(self._signal)


class CustomCheckBox(QCheckBox):
    """customized check box to return widget name and value when current check state is changed"""
    changed = pyqtSignal(str, bool)
```

```
    def __init__(self, wgt_name_='CustomCheckBox', *args, **kwargs):
        super(CustomCheckBox, self).__init__(*args, **kwargs)
        self._wgt_name = wgt_name_
        self.stateChanged.connect(self._event)

    def name(self):
        """return widget name"""
        return self._wgt_name

    def _event(self):
        self.changed.emit(self._wgt_name, self.checkState())


class CustomComboBox(QComboBox):
    """customized combo box to return widget name when current index is changed"""
    changed = pyqtSignal(str)

    def __init__(self, wgt_name_='CustomComboBox', *args, **kwargs):
        super(CustomComboBox, self).__init__(*args, **kwargs)
        self._wgt_name = wgt_name_
        self.currentIndexChanged.connect(self._event)

    def name(self):
        """return widget name"""
        return self._wgt_name

    def _event(self):
        self.changed.emit(self._wgt_name)


class CustomRadioButton(QRadioButton):
    """customized radio button to return widget name when check state is changed"""
    changed = pyqtSignal(str)

    def __init__(self, wgt_name_='CustomRadioButton', *args, **kwargs):
        super(CustomRadioButton, self).__init__(*args, **kwargs)
        self._wgt_name = wgt_name_
        self.toggled.connect(self._event)

    def name(self):
        """return widget name"""
        return self._wgt_name

    def _event(self):
        self.changed.emit(self._wgt_name)


class CustomMplCanvas(FigureCanvas):
    """DIY figure canvas"""
    def __init__(self, data_=None, parent_=None, width_=5, height_=4, dpi_=100):
        self._parent = parent_
```

```
        self._fig = Figure(figsize=(width_, height_), dpi=dpi_)
        self._axes = self._fig.add_subplot(111)
        self._plot_figure(data_)

        super(CustomMplCanvas, self).__init__(self._fig)
        self.setParent(parent_)
        FigureCanvas.setSizePolicy(self, QSizePolicy.Expanding, QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)
    #     self._tool_bar = NavigationToolbar(self, self._parent)

    # def tool_bar(self):
    #     """..."""
    #     return self._tool_bar

    def _plot_figure(self, data_):
        """plot figure using given data"""
        raise NotImplementedError("this method needs to be defined by subclass")


class CustomTableWidget(QTableWidget):
    """customized table widget to enable right click events"""
    rightClicked = pyqtSignal(int)

    def mousePressEvent(self, e):
        """..."""
        super(CustomTableWidget, self).mousePressEvent(e)
        if e.buttons() == Qt.RightButton:
            self.rightClicked.emit(self.currentRow())
```

- gui\help.py

```
# coding=utf-8
"""help doc"""

from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QDialog, QDialogButtonBox, QPlainTextEdit, QTabWidget,
QVBoxLayout

help_content = [
    ("Inst Params", """1. Strike - strike price of an OPTION

2. Qty - unit of each instrument
    * could be a FLOAT number
    * could be NEGATIVE indicating SHORT position

3. Premium - unit cost / premium of an OPTION"""),

    ("Curve Types", """From portfolio view:
1. Payoff Curve
    * portfolio payoff at maturity
2. PV Curve
    * portfolio current PV
3. Delta Curve
```

```
    * portfolio current Delta
4. Gamma Curve
    * portfolio current Gamma
    * Monte-Carlo is not recommended

From investment view:
1. Net Payoff Curve
    * portfolio payoff at maturity minus portfolio cost
2. PnL Curve
    * portfolio current PnL
    * portfolio PV minus portfolio cost"""),

    ("Pricing Tips", """1. Right click an OPTION for auto pricing
    * right click on the target line

2. Edit pricing env in Menu - Config - Pricing Env

3. Plotting for portfolios with STOCK may become confusing
    when dividend yield is not zero.
    Because of the difference between STOCK and FORWARD,
    STOCK cannot be used to hedge OPTION directly according
    to the DELTA curve."""),

    ("Pricing Params", """1. Annual Risk Free Rate (%, default 3)
2. Underlying Volatility (%, default 30)
3. Dividend Yield Ratio (%, default 0)
4. Portfolio Maturity (y)
5. Cost Rounding (default 2)
6. Rate Format (default Single)
    * Single or Compound (continuous)
    * if Single is chosen, 1 & 3 will shifted via:
    * r_c = (ln(1 + r / 100) - 1) * 100
7. Pricing Engine (default Black-Scholes)
    * Black-Scholes or Monte-Carlo""")
]


class HelpDialog(QDialog):
    """help doc dialog"""
    def __init__(self, parent_, *args, **kwargs):
        self._parent = parent_
        super(HelpDialog, self).__init__(*args, **kwargs)
        self.setAttribute(Qt.WA_DeleteOnClose)
        self.setWindowTitle("Help")
        # initialize basic widgets
        self._main_layout = QVBoxLayout(self)
        # setup and show
        self.setup_ui()
        self.setLayout(self._main_layout)
        self.show()

    def setup_ui(self):
```

```
        """setup all ui components"""
        _tab = QTabWidget()
        for _content in help_content:
            _wgt = QPlainTextEdit(_content[1])
            _wgt.setFocusPolicy(Qt.NoFocus)
            _tab.addTab(_wgt, _content[0])
        self._main_layout.addWidget(_tab)
        _btn = QDialogButtonBox(QDialogButtonBox.Ok)
        _btn.button(QDialogButtonBox.Ok).setDefault(True)
        _btn.accepted.connect(self.accept)
        self._main_layout.addWidget(_btn)
```

- gui\main.py

```
# coding=utf-8
"""
Vanilla Portfolio Ralated Curve Generator
Version 1.0.0

This is the group project of Chenxi Xiang, Peiwen Zhang and Tianyi Ji

A simple tool to estimate the different spot-based curves of vanilla portfolios.

Pricing is now available for vanilla options based on Black-Scholes or Monte-Carlo methods.
"""

from sys import path as sys_path
sys_path.append("{}/..".format(sys_path[0]))

from PyQt5.QtCore import QRect, Qt
from PyQt5.QtWidgets import QApplication, QFileDialog, QHBoxLayout, QMainWindow, QMenu,
QMessageBox, QPushButton
from PyQt5.QtWidgets import QVBoxLayout, QWidget
from gui.custom import CustomPushButton
from gui.help import HelpDialog
from gui.table import InstTable
from gui.plot import PayoffCurve, PlotParam
from gui.pricing_env import PricingEnv, parse_env
from instrument import Instrument
from instrument.default_param import env_default_param
from instrument.env_param import EngineMethod
from instrument.portfolio import CurveType, Portfolio
from json import dumps, loads
from numpy import array
from sys import argv as sys_argv, exit as sys_exit


btn_group = [
    [
        ("Payoff Curve", CurveType.Payoff.value),
        ("Net Payoff Curve", CurveType.NetPayoff.value),
        ("PnL Curve", CurveType.PnL.value),
    ],
```

```
        [
            ("PV Curve", CurveType.PV.value),
            ("Delta Curve", CurveType.Delta.value),
            ("Gamma Curve", CurveType.Gamma.value),
        ],
    ]

    MC_warning_curve = [CurveType.PnL.value, CurveType.PV.value, CurveType.Delta.value,
    CurveType.Gamma.value]


    class ApplicationWindow(QMainWindow):
        """
        application main window
        an instrument editor on the left
        a curve viewer on the right
        """
        def __init__(self):
            QMainWindow.__init__(self)
            # set basic parameters
            self.setAttribute(Qt.WA_DeleteOnClose)
            self.setWindowTitle("Option Portfolio Payoff Curve")
            # initialize basic widgets
            self._main = QWidget(self)
            self._plot = QWidget(self._main)
            self._table = QWidget(self._main)
            self._env_box = QWidget(self._main)
            self._help_box = QWidget(self._main)
            # initialize data storage
            self.env_data = env_default_param
            self._last_path = '.'
            # setup and show
            self.setup_ui()
            self.show()

        def setup_ui(self):
            """setup menu, option editor, and payoff curve viewer"""
            self._set_menu()
            self._plot = PayoffCurve(dict(x=array([]), y=array([]), type="Payoff"), self._main)
            self._set_table()

            _main_layout = QHBoxLayout(self._main)

            _vbox = QVBoxLayout()
            _vbox.setSpacing(0)
            _vbox.addWidget(self._table)
            _vbox.addLayout(self._inst_btn_layout())
            _main_layout.addLayout(_vbox)

            _vbox = QVBoxLayout()
            _vbox.setSpacing(0)
            _vbox.addWidget(self._plot)
```

```
        # _vbox.addWidget(self._plot.tool_bar())

        _sub_vbox = QVBoxLayout()
        _sub_vbox.setContentsMargins(0, 8, 0, 0)
        _sub_vbox.setSpacing(0)
        for _btn in btn_group:
            _sub_vbox.addLayout(self._plot_btn_layout(_btn))
        _vbox.addLayout(_sub_vbox)

        _main_layout.addLayout(_vbox)
        self._main.setFocus()
        self.setCentralWidget(self._main)
        _width, _height = self._get_width_height()
        self.setGeometry(QRect(100, 100, _width, _height))

    def _get_width_height(self):
        _plot_width, _plot_height = self._plot.get_width_height()
        return 112 + self._table.col_width() + _plot_width, 200 + _plot_height

    def _load(self):
        _file_path, _file_type = QFileDialog.getOpenFileName(
            self, "Load Portfolio", self._last_path, "JSON Files (*.json)")
        if not _file_path:
            return

        with open(_file_path) as f:
            _input_data = loads(f.read())
        self._last_path = _file_path

        _raw_data = _input_data.get('data')
        _env = _input_data.get('env')

        if _raw_data and _env:
            self.env_data = _env
            try:
                while self._table.rowCount():
                    self._table.removeRow(0)

                for _row in _raw_data:
                    self._add(_row)

            except Exception as e:
                QMessageBox.warning(self, "Load Portfolio", "Invalid data in {}\nError Message:{}".format(
                    _file_path, str(e)))

        else:
            QMessageBox.warning(self, "Load Portfolio", "No data found in {}".format(_file_path))

    def _save(self):
        _raw_data = self._collect()
        _output = dict(data=_raw_data, env=self.env_data)
```

```
            if _raw_data:
                _file_path, _file_type = QFileDialog.getSaveFileName(
                    self, "Save Portfolio", self._last_path, "JSON Files (*.json)")
                if not _file_path:
                    return

                with open(_file_path, 'w') as f:
                    f.write(dumps(_output, indent=4))
                self._last_path = _file_path

        def _export(self):
            _file_path, _file_type = QFileDialog.getSaveFileName(
                self, "Save Portfolio", self._last_path, "PNG Files (*.png)")
            if not _file_path:
                return

            self._plot.save(_file_path)

        def _pricing_env(self):
            self._env_box = PricingEnv(self)

        def _about(self):
            QMessageBox.about(self, "About", __doc__)

        def _help(self):
            self._help_box = HelpDialog(self)

        def _quit(self):
            self.close()

        def closeEvent(self, ce):
            """event when close button is clicked"""
            self._quit()

        def _set_menu(self):
            self._menu = self.menuBar()
            self._menu.setNativeMenuBar(False)

            _file = QMenu("&File", self)
            _file.addAction("&Load", self._load, Qt.CTRL + Qt.Key_L)
            _file.addAction("&Save", self._save, Qt.CTRL + Qt.Key_S)
            _file.addAction("&Export", self._export, Qt.CTRL + Qt.Key_E)
            _file.addAction("&Quit", self._quit, Qt.CTRL + Qt.Key_Q)
            self._menu.addMenu(_file)

            _config = QMenu("&Config", self)
            _config.addAction("&Pricing Env", self._pricing_env, Qt.CTRL + Qt.Key_P)
            self._menu.addMenu(_config)

            _help = QMenu("&Help", self)
            _help.addAction("&Help", self._help, Qt.CTRL + Qt.Key_H)
            _help.addAction("&About", self._about, Qt.CTRL + Qt.Key_A)
```

```
        self._menu.addMenu(_help)

    def _inst_btn_layout(self):
        _hbox = QHBoxLayout()

        _add_btn = QPushButton("Add")
        _add_btn.clicked.connect(self._add)
        _hbox.addWidget(_add_btn)

        _copy_btn = QPushButton("Copy")
        _copy_btn.clicked.connect(self._copy)
        _hbox.addWidget(_copy_btn)

        _delete_btn = QPushButton("Delete")
        _delete_btn.clicked.connect(self._delete)
        _hbox.addWidget(_delete_btn)

        return _hbox

    def _plot_btn_layout(self, btn_group_):
        _hbox = QHBoxLayout()
        for _btn in btn_group_:
            _plot_btn = CustomPushButton(display_=_btn[0], signal_=_btn[1])
            _plot_btn.pressed.connect(self._plot_impl)
            _hbox.addWidget(_plot_btn)
        return _hbox

    def _set_table(self):
        self._table = InstTable(self)
        self._add()
        self._plot_payoff()

    def _add(self, data_=None):
        try:
            self._table.add_row(data_)
        except Exception as e:
            QMessageBox.warning(
                self, "Add Instrument", "An error occurred while adding new instrument: {}".format(str(e)))

    def _copy(self):
        self._table.copy_row()

    def _delete(self):
        self._table.delete_row()

    def _collect(self):
        return self._table.collect()

    def _prepare_data(self):
        _raw_data = self._table.collect()
        _inst = [Instrument.get_inst(_data) for _data in _raw_data] if _raw_data else []
        _inst_show = [Instrument.get_inst(_data)
```

```
                for _data in filter(lambda x: x[PlotParam.Show.value], _raw_data)] if _raw_data else []
        _portfolio = Portfolio(_inst)
        _mkt, _engine, _rounding = parse_env(self.env_data)
        _portfolio.set_mkt(_mkt)
        _portfolio.set_engine(_engine)
        _portfolio.set_show(_inst_show)
        return _portfolio

    def _plot_payoff(self):
        self._plot_impl(CurveType.Payoff.value)

    def _plot_net_payoff(self):
        self._plot_impl(CurveType.NetPayoff.value)

    def _plot_pnl(self):
        self._plot_impl(CurveType.PnL.value)

    def _plot_pv(self):
        self._plot_impl(CurveType.PV.value)

    def _plot_delta(self):
        self._plot_impl(CurveType.Delta.value)

    def _plot_impl(self, type_):
        _portfolio = self._prepare_data()
        if _portfolio.engine['engine'] == EngineMethod.MC.value and type_ in MC_warning_curve:
            if QMessageBox.question(
                    self, "Evaluation Cure",
                    "Using Monte-Carlo to generate Evaluation Curve might be extremely time consuming. "
                    "Are you sure to continue?") == QMessageBox.No:
                return
        _x, _y = _portfolio.gen_curve(type_, full_=True)
        _x_ref = 0 if type_ == CurveType.PnL.value else 100 if _portfolio.has_stock() else 0
        self._plot.update_figure(dict(x=_x, y=_y, type=type_, x_ref=_x_ref, y_ref=_portfolio.center()))

    def _test(self):
        pass


if __name__ == '__main__':
    app = QApplication(sys_argv)
    main = ApplicationWindow()
    sys_exit(app.exec_())
```

- gui\plot.py

```
# coding=utf-8
"""plotting template"""

from enum import Enum
from gui.custom import CustomMplCanvas
from numpy import array, zeros
from utils import PRECISION_ZERO
```

```
class PlotParam(Enum):
    """plotting parameters"""
    Show = 'Show'


plot_default_param = {
    PlotParam.Show.value: False,
}


class PayoffCurve(CustomMplCanvas):
    """figure canvas for plotting payoff curve"""

    def _plot_figure(self, data_):
        """
        plot payoff curve using given data
        :param data_: a dict consists with x (numpy array) and y (numpy array) in same dimension
        """
        _x = data_.get('x', array([]))
        _y = array(data_.get('y', [array([])]))
        _type = data_.get('type')
        _x_ref = data_.get('x_ref', 0)
        _y_ref = data_.get('y_ref', 100)

        if not _type:
            raise ValueError("plot type is required")

        if _x.size and _y.size:
            self._axes.clear()
            self._axes.plot((_y_ref, _y_ref), (_y.min(), _y.max()), color="grey", linewidth=1.5)

            if _y.min() <= _x_ref <= _y.max() \
                    or abs(_y.min() - _x_ref) <= PRECISION_ZERO or abs(_y.max() - _x_ref) <=
PRECISION_ZERO:
                self._axes.plot(_x, zeros(_x.size) + _x_ref, color="grey", linewidth=1.5)

            if len(_y) > 1:
                for _line in _y[1:]:
                    self._axes.plot(_x, _line, color="blue", linestyle='--')
            self._axes.plot(_x, _y[0], color="red", linestyle='-')

        self._set_axis(_type)

    def update_figure(self, data_):
        """
        update payoff curve using new data
        :param data_: a dict consists with x (numpy array) and y (list of numpy array)
            each array should be in same dimension
        """
        self._plot_figure(data_)
```

```
        self.draw()

    def save(self, file_path_):
        """
        save figure to file using given path
        :param file_path_: a str indicating path to save figure file
        """
        self.print_png(file_path_)

    def _set_axis(self, type_):
        self._axes.set_xlabel("Spot")
        # self._axes.set_ylabel(type_)
        self._axes.set_title("Option Portfolio {} Curve".format(type_))
        self._axes.grid(axis='x', linewidth=0.75, linestyle='-', color='0.75')
        self._axes.grid(axis='y', linewidth=0.75, linestyle='-', color='0.75')
```

- gui\pricing_env.py

```
# coding=utf-8
"""pricing env dialog"""

from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QButtonGroup, QDialog, QDialogButtonBox, QHBoxLayout, QLabel,
QVBoxLayout, QLineEdit
from copy import deepcopy
from enum import Enum
from gui.custom import CustomRadioButton
from instrument.default_param import env_default_param
from instrument.env_param import EngineMethod, EngineParam, EnvParam, RateFormat
from utils import float_int


class FieldType(Enum):
    """Field type"""
    String = 0
    Number = 1
    Radio = 2


fixed_width = 180

env_param = [
    (FieldType.Number.value, EnvParam.RiskFreeRate.value, "Annual Risk Free Rate (%):",
fixed_width,
     None, None, None),
    (FieldType.Number.value, EnvParam.UdVolatility.value, "Ud Volatility (%):", fixed_width,
     None, None, None),
    (FieldType.Number.value, EnvParam.UdDivYieldRatio.value, "Ud Dividend Yield (%):",
fixed_width,
     None, None, None),
    (FieldType.Number.value, EnvParam.UdSpotForPrice.value, "Ud Spot for Pricing:", fixed_width,
     None, None, None),
    (FieldType.Number.value, EnvParam.PortMaturity.value, "Time to Maturity (Y):", fixed_width,
```

```
        None, None, None),
        (FieldType.Number.value, EnvParam.CostRounding.value, "Instrument Cost Rounding:",
    fixed_width,
        None, None, None),
        (FieldType.Radio.value, EnvParam.RateFormat.value, "Input Rate Format:", fixed_width,
         [_r.value for _r in RateFormat], None, None),
        (FieldType.Radio.value, EnvParam.PricingEngine.value, "Pricing Engine:", fixed_width,
         [_e.value for _e in EngineMethod], None, None),
        (FieldType.Number.value, EngineParam.MCIteration.value, "Monte-Carlo Iterations:",
    fixed_width,
        None, EnvParam.PricingEngine.value, EngineMethod.MC.value),
    ]


    class PricingEnv(QDialog):
        """
        dialog for editing pricing environment parameters
        included paramters should be all defined above - env_param
        """
        def __init__(self, parent_, *args, **kwargs):
            super(PricingEnv, self).__init__(*args, **kwargs)
            self._parent = parent_
            self.setAttribute(Qt.WA_DeleteOnClose)
            self.setWindowTitle("Pricing Env")
            # initialize basic widgets
            self._main_layout = QVBoxLayout(self)
            # setup and show
            self.setup_ui()
            self.setLayout(self._main_layout)
            self.show()

        def setup_ui(self):
            """setup all parameter input widget and buttons"""
            for _param in env_param:
                self._add_param(_param)
            _btn = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel |
    QDialogButtonBox.Reset)
            _btn.button(QDialogButtonBox.Ok).autoDefault()
            _btn.button(QDialogButtonBox.Reset).clicked.connect(self._on_reset)
            _btn.accepted.connect(self._on_ok)
            _btn.rejected.connect(self.reject)
            self._main_layout.addWidget(_btn)

        def _add_param(self, param_):
            if param_[0] in [FieldType.String.value, FieldType.Number.value]:
                _hbox = QHBoxLayout()
                _label = QLabel(param_[2])
                _label.setFixedWidth(param_[3])
                _hbox.addWidget(_label)
                _wgt = QLineEdit(self)
                _wgt.setAlignment(Qt.AlignRight)
                _default = self._parent.env_data.get(param_[1])
```

```
            if _default is not None:
                _wgt.setText(str(_default))
            self.__setattr__(param_[1], _wgt)
            _hbox.addWidget(_wgt)
            self._main_layout.addLayout(_hbox)

            if param_[5] is not None:
                try:
                    _grand_parent = self.__getattribute__(param_[5])
                    for _btn in _grand_parent.buttons():
                        _btn.changed.connect(self._radio_connection)
                        if not hasattr(_btn, 'param'):
                            _btn.__setattr__('param', [])

                    _parent = self.__getattribute__(param_[6])
                    _parent.param.append(param_[1])
                    _parent.__setattr__(param_[1], _wgt)
                    _wgt.setEnabled(_parent.isChecked())

                except AttributeError as e:
                    raise Exception(str(e))

        elif param_[0] == FieldType.Radio.value:
            _vbox = QVBoxLayout()
            _label = QLabel(param_[2])
            _label.setFixedWidth(param_[3])
            _vbox.addWidget(_label)
            _hbox = QHBoxLayout()
            _btn_group = QButtonGroup()
            self.__setattr__(param_[1], _btn_group)
            _range = param_[4]
            for _idx, _item in enumerate(_range):
                _wgt = CustomRadioButton(_item, _item, self)
                self.__setattr__(_item, _wgt)
                _hbox.addWidget(_wgt)
                _btn_group.addButton(_wgt, _idx)
            _vbox.addLayout(_hbox)
            self._main_layout.addLayout(_vbox)
            _default = self._parent.env_data.get(param_[1])
            self.__getattribute__(_default).setChecked(True)

    def _radio_connection(self, wgt_name_):
        _wgt = self.__getattribute__(wgt_name_)
        for _param in _wgt.param:
            _child = _wgt.__getattribute__(_param)
            _child.setEnabled(_wgt.isChecked())

    def _on_ok(self):
        _env = dict()
        for _param in env_param:
            _env[_param[1]] = self._get_wgt_value(_param[1], _param[0], _param[4])
```

```
            self._parent.env_data = _env
            self.accept()

        def _on_reset(self):
            for _param in env_param:
                self._set_wgt_value(_param[1], _param[0], env_default_param[_param[1]])

        def _set_wgt_value(self, wgt_name_, wgt_type_, value_):
            _wgt = self.__getattribute__(wgt_name_)
            if wgt_type_ in [FieldType.String.value, FieldType.Number.value]:
                _wgt.setText(str(value_))
            elif wgt_type_ == FieldType.Radio.value:
                for _btn in _wgt.buttons():
                    if _btn.name() == value_:
                        _btn.setChecked(True)
            else:
                raise ValueError("invalid widget type {}".format(wgt_type_))

        def _get_wgt_value(self, wgt_name_, wgt_type_, *args):
            _wgt = self.__getattribute__(wgt_name_)
            if wgt_type_ == FieldType.String.value:
                return _wgt.text()
            elif wgt_type_ == FieldType.Number.value:
                return float_int(_wgt.text())
            elif wgt_type_ == FieldType.Radio.value:
                _range = args[0]
                return _range[_wgt.checkedId()]
            else:
                return None


    def parse_env(env_param_):
        """parse environment data into market, engine, and rounding"""
        _mkt = deepcopy(env_param_)
        _engine = dict(engine=_mkt.pop(EnvParam.PricingEngine.value), param={})
        for _engine_param in [_param for _param in env_param if _param[5] ==
    EnvParam.PricingEngine.value]:
            _engine['param'][_engine_param[1]] = _mkt.pop(_engine_param[1])
        _rounding = _mkt.pop(EnvParam.CostRounding.value)
        return _mkt, _engine, _rounding
```

- gui\table.py

```
# coding=utf-8
"""instrument table template"""

from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QAbstractItemView, QMessageBox, QTableWidgetItem
from enum import Enum
from gui.custom import CustomCheckBox, CustomComboBox, CustomTableWidget
from gui.plot import PlotParam
from gui.pricing_env import parse_env
from instrument import InstType, InstParam, Instrument, option_type
```

```
from instrument.default_param import default_param, default_type
from instrument.env_param import EnvParam
from utils import float_int


class TableCol(Enum):
    """table column"""
    Type = 'Type'
    Strike = 'Strike'
    Maturity = 'Maturity'
    Qty = 'Qty'
    Premium = 'Premium'
    Show = 'Show'


class ColType(Enum):
    """column type"""
    String = 0
    Number = 1
    Boolean = 2
    Other = 3


table_col = [
    (TableCol.Type.value, ColType.Other.value, "Type", InstParam.InstType.value, 80),
    (TableCol.Strike.value, ColType.Number.value, "Strike", InstParam.OptionStrike.value, 50),
    (TableCol.Qty.value, ColType.Number.value, "Qty", InstParam.InstUnit.value, 50),
    (TableCol.Premium.value, ColType.Number.value, "Premium", InstParam.InstCost.value, 60),
    (TableCol.Show.value, ColType.Boolean.value, "", PlotParam.Show.value, 30),
]


class InstTable(CustomTableWidget):
    """
    instrument table widget to edit instrument info
    all table columns should be defined above - table_col
    """
    _seq = 0

    def __init__(self, parent_, *args, **kwargs):
        super(InstTable, self).__init__(0, len(table_col), *args, **kwargs)
        self._parent = parent_
        self.setHorizontalHeaderLabels([_col[2] for _col in table_col])
        for _idx, _col in enumerate(table_col):
            self.setColumnWidth(_idx, _col[4])
        self._col_width = sum([_col[4] for _col in table_col])
        self.setSelectionBehavior(QAbstractItemView.SelectRows)
        self.setSelectionMode(QAbstractItemView.SingleSelection)
        self.rightClicked.connect(self._price)

    def col_width(self):
        """return width sum of all columns"""
```

```
        return self._col_width

    def add_row(self, data_=None):
        """add a new instrument with given or default data"""
        self.setRowCount(self.rowCount() + 1)
        _id = self._inst_id()
        _type = data_.get(InstParam.InstType.value, default_type) if data_ else default_type

        for _idx, _col in enumerate(table_col):
            if _col[1] in [ColType.String.value, ColType.Number.value]:
                _default = default_param[_type].get(_col[3], '-')
                if _default == EnvParam.UdSpotForPrice.value:
                    _default = self._parent.env_data.get(EnvParam.UdSpotForPrice.value, '-')
                _content = data_.get(_col[3], _default) if data_ else _default
                _wgt = QTableWidgetItem(str(_content))
                _wgt.setTextAlignment(Qt.AlignCenter)
                self.setItem(self.rowCount() - 1, _idx, _wgt)

            elif _col[1] == ColType.Boolean.value:
                _default = default_param[_type].get(_col[3], False)
                _content = data_.get(_col[3], _default) if data_ else _default
                _wgt = QTableWidgetItem()
                _wgt.setCheckState(Qt.Checked if _content else Qt.Unchecked)
                self.setItem(self.rowCount() - 1, _idx, _wgt)

            elif _col[1] == ColType.Other.value:
                if _col[0] == TableCol.Type.value:
                    _wgt_name = '{}_type'.format(_id)
                    _wgt = QTableWidgetItem(_wgt_name)
                    _wgt._wgt = CustomComboBox(wgt_name_=_wgt_name)
                    for _inst_type in [_t.value for _t in InstType]:
                        _wgt._wgt.addItem(_inst_type)
                    _wgt._wgt.setCurrentText(_type)
                    _wgt._wgt.setFixedWidth(_col[4])
                    self.__setattr__(_wgt_name, _wgt._wgt)
                    _wgt._wgt.changed.connect(self._set_default)
                    _wgt.setTextAlignment(Qt.AlignCenter)
                    self.setItem(self.rowCount() - 1, _idx, _wgt)
                    self.setCellWidget(self.rowCount() - 1, _idx, _wgt._wgt)
                else:
                    raise ValueError("invalid table column '{}'".format(_col[0]))

            else:
                raise ValueError("invalid column type '{}'".format(_col[1]))

    def copy_row(self):
        """copy an existing instrument and create a new one"""
        self.add_row()
        _row = self.currentRow()
        _raw_data = self._collect_row(_row)

        for _idx, _col in enumerate(table_col):
```

```
            if _col[1] in [ColType.String.value, ColType.Number.value]:
                self.item(self.rowCount() - 1, _idx).setText(str(_raw_data[_col[3]]))

            elif _col[1] == ColType.Boolean.value:
                self.item(self.rowCount() - 1, _idx).setCheckState(Qt.Checked if _raw_data[_col[3]] else
    Qt.Unchecked)

            elif _col[1] == ColType.Other.value:
                if _col[0] == TableCol.Type.value:
                    self.__getattribute__(
                        self.item(self.rowCount() - 1, _idx).text()).setCurrentText(_raw_data[_col[3]])
                else:
                    raise ValueError("invalid table column '{}'".format(_col[0]))

        def delete_row(self):
            """delete an instrument"""
            if self.rowCount() == 1:
                QMessageBox.information(self, "Warning", "Only one option left, cannot be deleted.")
            else:
                _row = self.currentRow()
                self.removeRow(_row)

        def collect(self):
            """collect all instruments data"""
            return [self._collect_row_full(_row) for _row in range(self.rowCount())]

        def _collect_row_full(self, row_):
            _data_dict = self._collect_row(row_)
            _type = _data_dict.get(InstParam.InstType.value)
            if _type in option_type:
                _data_dict[InstParam.OptionMaturity.value] =
    self._parent.env_data[EnvParam.PortMaturity.value]
            return _data_dict

        def _collect_row(self, row_):
            _data_dict = dict()
            for _idx, _col in enumerate(table_col):
                if _col[1] == ColType.String.value:
                    _data = self.item(row_, _idx).text()
                    _data_dict[_col[3]] = _data
                elif _col[1] == ColType.Number.value:
                    _data = float_int(self.item(row_, _idx).text())
                    _data_dict[_col[3]] = _data
                elif _col[1] == ColType.Boolean.value:
                    _data = self.item(row_, _idx).checkState() == Qt.Checked
                    _data_dict[_col[3]] = _data
                elif _col[1] == ColType.Other.value:
                    if _col[0] == TableCol.Type.value:
                        _data = self.__getattribute__(self.item(row_, _idx).text()).currentText()
                        _data_dict[_col[3]] = _data
                    else:
                        raise ValueError("invalid table column '{}'".format(_col[0]))
```

```
        return _data_dict

    def _set_default(self, wgt_name_):
        _type = None
        for _row in range(self.rowCount()):
            for _idx, _col in enumerate(table_col):
                if _col[0] == TableCol.Type.value and self.item(_row, _idx).text() == wgt_name_:
                    _type = self.__getattribute__(self.item(_row, _idx).text()).currentText()
                    break

            if _type:
                for _idx, _col in enumerate(table_col):
                    if _col[1] in [ColType.String.value, ColType.Number.value]:
                        _default = default_param[_type].get(_col[3], '-')
                        if _default == EnvParam.UdSpotForPrice.value:
                            _default = self._parent.env_data.get(EnvParam.UdSpotForPrice.value, '-')
                        self.item(_row, _idx).setText(str(_default))
                        self.item(_row, _idx).setFlags(Qt.ItemIsEnabled | Qt.ItemIsEditable |
Qt.ItemIsSelectable)
                    elif _col[1] == ColType.Boolean.value:
                        _default = default_param[_type].get(_col[3], False)
                        self.item(_row, _idx).setCheckState(Qt.Checked if _default else Qt.Unchecked)
                        self.item(_row, _idx).setFlags(Qt.ItemIsEnabled | Qt.ItemIsEditable | Qt.ItemIsSelectable
|
                                                       Qt.ItemIsUserCheckable)
                    elif _col[1] == ColType.Other.value:
                        pass

                if _type == InstType.Stock.value:
                    for _idx, _col in enumerate(table_col):
                        if _col[3] in [InstParam.OptionStrike.value]:
                            self.item(_row, _idx).setText('-')
                            self.item(_row, _idx).setFlags(Qt.ItemIsSelectable)
                return
        raise ValueError("missing default value of {}".format(wgt_name_))

    def _set_header(self):
        for _idx, _col in enumerate(table_col):
            _wgt = QTableWidgetItem(_col[2])
            if _col[1] in [ColType.String.value, ColType.Number.value]:
                pass
            elif _col[1] == ColType.Boolean.value:
                _check = CustomCheckBox(str(_idx))
                _check.setCheckState(Qt.Unchecked)
                _check.changed.connect(self._on_check_all)
            elif _col[1] == ColType.Other.value:
                if _col[0] == TableCol.Type.value:
                    pass
                else:
                    raise ValueError()
            else:
                raise ValueError()
```

```
            self.setHorizontalHeaderItem(_idx, _wgt)

        def _on_check_all(self, wgt_name_, check_state_):
            _idx = int(wgt_name_)
            for _row in range(self.rowCount()):
                self.item(_row, _idx).setCheckState(check_state_)

        def _price(self, row_):
            if row_ == -1:
                return
            # prepare instrument data
            _raw_data = self._collect_row_full(row_)
            # prepare pricing environment
            _mkt, _engine, _rounding = parse_env(self._parent.env_data)
            # do pricing
            _inst = Instrument.get_inst(_raw_data)
            _price = _inst.pv(_mkt, _engine, unit_=1)
            for _idx, _col in enumerate(table_col):
                if _col[0] == TableCol.Premium.value:
                    self.item(row_, _idx).setText(str(round(_price, _rounding)))

        def _inst_id(self):
            self._seq += 1
            return "Inst-{}".format(self._seq)
```

- instrument__init__.py

```
# coding=utf-8
"""definition of base instrument"""

from enum import Enum
from numpy.ma import exp
from instrument.env_param import EngineMethod, EngineParam, EnvParam, RateFormat
from utils import to_continuous_rate


class InstParam(Enum):
    """instrument parameters"""
    InstISP = 'InstISP'
    InstType = 'InstType'
    InstUnit = 'InstUnit'
    InstCost = 'InstCost'
    OptionType = 'OptionType'
    OptionStrike = 'OptionStrike'
    OptionMaturity = 'OptionMaturity'


class InstType(Enum):
    """instrument type"""
    CallOption = 'CALL'
    PutOption = 'PUT'
    Stock = 'STOCK'
```

```
option_type = [InstType.CallOption.value, InstType.PutOption.value]


class Instrument(object):
    """
    financial instrument class
    use class method - get_inst to get correct type of instrument
    """
    _name = "instrument"
    _inst_dict = None
    _type = None
    _unit = None
    _price = None

    def __init__(self, inst_dict_):
        self._inst_dict = inst_dict_
        self.type = inst_dict_.get(InstParam.InstType.value)
        self.unit = inst_dict_.get(InstParam.InstUnit.value)
        self.price = inst_dict_.get(InstParam.InstCost.value)

    def __str__(self):
        return "{} * {}".format(self.unit, self.type)

    @classmethod
    def get_inst(cls, inst_dict_):
        """get instrument through instrument dictionary"""
        type_ = inst_dict_.get(InstParam.InstType.value)
        if type_ in option_type:
            from instrument.option import Option
            return Option(inst_dict_)
        elif type_ == InstType.Stock.value:
            from instrument.stock import Stock
            return Stock(inst_dict_)
        if type_ is None:
            raise ValueError("instrument type not specified")

    def payoff(self, mkt_dict_):
        """get instrument payoff for given spot"""
        raise NotImplementedError("'payoff' method need to be defined in sub-classes")

    def net_payoff(self, mkt_dict_):
        """get instrument net payoff for given spot"""
        return self.payoff(mkt_dict_) - self.unit * self.price

    def profit_discount(self, mkt_dict_, time_):
        """get instrument pnl for given spot"""
        _rate, _spot = tuple(self._load_market(mkt_dict_, [EnvParam.RiskFreeRate.value,
EnvParam.UdSpotForPrice.value]))
        return self.payoff(_spot) * exp(-_rate * time_) - self.unit * self.price

    def pnl(self, mkt_dict_, engine_):
```

```
        """get instrument pnl for given spot"""
        return (self.pv(mkt_dict_, engine_, unit_=1) - self.price) * self.unit

    def pv(self, mkt_dict_, engine_, unit_=None):
        """evaluate instrument PV on given market"""
        raise NotImplementedError("'pv' method need to be defined in sub-classes")

    def delta(self, mkt_dict_, engine_, unit_=None):
        """evaluate instrument DELTA with market data and engine"""
        raise NotImplementedError("'delta' method need to be defined in sub-classes")

    def gamma(self, mkt_dict_, engine_, unit_=None):
        """evaluate instrument GAMMA with market data and engine"""
        raise NotImplementedError("'gamma' method need to be defined in sub-classes")

    @property
    def type(self):
        """instrument type"""
        if self._type is None:
            raise ValueError("{} type not specified".format(self._name))
        return self._type

    @type.setter
    def type(self, type_):
        if type_ not in [_type.value for _type in InstType]:
            raise ValueError("invalid {} type given".format(self._name))
        self._type = type_

    @property
    def unit(self):
        """instrument unit - number of instrument"""
        if self._unit is None:
            raise ValueError("{} unit not specified".format(self._name))
        return self._unit

    @unit.setter
    def unit(self, unit_):
        if unit_ is not None:
            if not isinstance(unit_, (int, float)):
                raise ValueError("type <int> is required for unit, not {}".format(type(unit_)))
            self._unit = unit_

    @property
    def price(self):
        """instrument price"""
        if self._price is None:
            raise ValueError("{} price not specified".format(self._name))
        return self._price

    @price.setter
    def price(self, price_):
        if price_ is not None:
```

```
            if not isinstance(price_, (int, float)):
                raise ValueError("type <int> or <float> is required for price, not {}".format(type(price_)))
            self._price = price_

    @staticmethod
    def _load_market(mkt_dict_, load_param_):
        _res = []
        for _param in load_param_:
            _value = mkt_dict_.get(_param)
            if _param in [EnvParam.RiskFreeRate.value, EnvParam.UdVolatility.value,
EnvParam.UdDivYieldRatio.value]:
                if not isinstance(_value, (int, float)):
                    raise ValueError("type <int> or <float> is required for {}, not {}".format(_param,
type(_value)))
                _value /= 100
            if _param in [EnvParam.RiskFreeRate.value, EnvParam.UdDivYieldRatio.value]:
                _rate_format = mkt_dict_.get(EnvParam.RateFormat.value)
                if _rate_format not in [_r.value for _r in RateFormat]:
                    raise ValueError("invalid rate type given: {}".format(_rate_format))
                if _rate_format == RateFormat.Single.value:
                    _value = to_continuous_rate(_value)
            _res.append(_value)
        return _res
```

- instrument\default_param.py

```
# coding=utf-8
"""default value of all parameters"""

from gui.plot import PlotParam
from instrument import InstParam, InstType
from instrument.env_param import EnvParam, EngineMethod, EngineParam, RateFormat


default_param = {
    InstType.CallOption.value: {
        InstParam.InstUnit.value: 1,
        InstParam.InstCost.value: 0,
        InstParam.OptionStrike.value: EnvParam.UdSpotForPrice.value,
        PlotParam.Show.value: False,
    },
    InstType.PutOption.value: {
        InstParam.InstUnit.value: 1,
        InstParam.InstCost.value: 0,
        InstParam.OptionStrike.value: EnvParam.UdSpotForPrice.value,
        PlotParam.Show.value: False,
    },
    InstType.Stock.value: {
        InstParam.InstUnit.value: 1,
        InstParam.InstCost.value: EnvParam.UdSpotForPrice.value,
        PlotParam.Show.value: False,
    }
}
```

```
default_type = InstType.CallOption.value

env_default_param = {
    EnvParam.RiskFreeRate.value: 3,
    EnvParam.UdVolatility.value: 30,
    EnvParam.UdDivYieldRatio.value: 0,
    EnvParam.UdSpotForPrice.value: 100,
    EnvParam.PortMaturity.value: 1,
    EnvParam.CostRounding.value: 2,
    EnvParam.RateFormat.value: RateFormat.Single.value,
    EnvParam.PricingEngine.value: EngineMethod.BS.value,
    EngineParam.MCIteration.value: 1000000,
}
```

- instrument\env_param.py

```
# coding=utf-8
"""market and engine parameters"""

from enum import Enum


class EnvParam(Enum):
    """market parameter"""
    RiskFreeRate = 'RiskFreeRate'
    UdVolatility = 'UdVolatility'
    UdDivYieldRatio = 'UdDivYieldRatio'
    UdSpotForPrice = 'UdSpotForPrice'
    PortMaturity = 'PortMaturity'
    CostRounding = 'CostRounding'
    RateFormat = 'RateFormat'
    PricingEngine = 'PricingEngine'


class RateFormat(Enum):
    """Rate format - single or continuously compounded"""
    Single = 'Single'
    Compound = 'Compound'


class EngineMethod(Enum):
    """engine evaluation method"""
    BS = 'Black-Scholes'
    MC = 'Monte-Carlo'


class EngineParam(Enum):
    """engine parameter"""
    MCIteration = 'MCIteration'
```

- instrument\option.py

OptionPayOffer
Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

```
# coding=utf-8
"""definition of option for payoff estimation and pricing"""

from instrument import InstParam, InstType, Instrument, option_type
from instrument.env_param import EngineMethod, EngineParam, EnvParam
from numpy import average, pi
from numpy.ma import exp, log, sqrt
from scipy.stats import norm


class Option(Instrument):
    """
    option class with basic parameters
    only vanilla option is available (barrier is not supported)
    can estimate option payoff under different level of spot
    can evaluate option price under different market using different evaluation engine
    """
    _name = "option"
    _strike = None
    _maturity = None

    def __init__(self, inst_dict_):
        super(Option, self).__init__(inst_dict_)
        self.strike = inst_dict_.get(InstParam.OptionStrike.value)
        self.maturity = inst_dict_.get(InstParam.OptionMaturity.value)

    def __str__(self):
        return "{} * {} {}, Maturity {}".format(self.unit, self.strike, self.type, self.maturity)

    def payoff(self, mkt_dict_):
        """get option payoff for given spot"""
        _spot = self._load_market(mkt_dict_, [EnvParam.UdSpotForPrice.value])[0]
        _reference = _spot - self.strike if self.type == InstType.CallOption.value else self.strike - _spot
        return max([_reference, 0]) * self.unit

    def pv(self, mkt_dict_, engine_, unit_=None):
        """calculate option PV with market data and engine"""
        _rate, _spot, _vol, _div, _method, _param, _sign, _strike, _t = self._prepare_risk_data(mkt_dict_,
engine_)
        _unit = unit_ or self.unit

        if _method == EngineMethod.BS.value:
            _d1 = (log(_spot / _strike) + (_rate - _div + _vol ** 2 / 2) * _t) / _vol / sqrt(_t)
            _d2 = _d1 - _vol * sqrt(_t)
            return _sign * (_spot * exp(-_div * _t) * norm.cdf(_sign * _d1) -
                            _strike * exp(-_rate * _t) * norm.cdf(_sign * _d2)) * _unit

        elif _method == EngineMethod.MC.value:
            from utils.monte_carlo import MonteCarlo
            _iteration = _param.get(EngineParam.MCIteration.value)
            if not _iteration:
                raise ValueError("iteration not specified")
```

```
            if not isinstance(_iteration, int):
                raise ValueError("type <int> is required for iteration, not {}".format(type(_iteration)))
            _spot = MonteCarlo.stock_price(_iteration, isp=_spot, rate=_rate, div=_div, vol=_vol, t=_t)
            _price = [max(_sign * (_s - _strike), 0) for _s in _spot]
            return average(_price) * exp(-_rate * _t) * _unit

    def delta(self, mkt_dict_, engine_, unit_=None):
        """calculate option DELTA with market data and engine"""
        _rate, _spot, _vol, _div, _method, _param, _sign, _strike, _t = self._prepare_risk_data(mkt_dict_,
engine_)
        _unit = unit_ or self.unit

        if _method == EngineMethod.BS.value:
            _d1 = (log(_spot / _strike) + (_rate + _vol ** 2 / 2) * _t) / _vol / sqrt(_t)
            return _sign * norm.cdf(_sign * _d1) * exp(-_div * _t) * _unit

        elif _method == EngineMethod.MC.value:
            from utils.monte_carlo import MonteCarlo
            _iteration = _param.get(EngineParam.MCIteration.value)
            if not _iteration:
                raise ValueError("iteration not specified")
            if not isinstance(_iteration, int):
                raise ValueError("type <int> is required for iteration, not {}".format(type(_iteration)))
            _spot = MonteCarlo.stock_price(_iteration, isp=_spot, rate=_rate, div=_div, vol=_vol, t=_t)
            _step = 0.01
            _delta = [(max(_sign * (_s + _step - _strike), 0) - max(_sign * (_s - _step - _strike), 0)) /
                      (_step * 2) for _s in _spot]
            return average(_delta) * exp(-_rate * _t) * _unit

    def gamma(self, mkt_dict_, engine_, unit_=None):
        """calculate option GAMMA with market data and engine"""
        _rate, _spot, _vol, _div, _method, _param, _sign, _strike, _t = self._prepare_risk_data(mkt_dict_,
engine_)
        _unit = unit_ or self.unit

        if _method == EngineMethod.BS.value:
            _d1 = (log(_spot / _strike) + (_rate + _vol ** 2 / 2) * _t) / _vol / sqrt(_t)
            return exp(-_d1 ** 2 / 2) / sqrt(2 * pi) / _spot / _vol / sqrt(_t) * exp(-_div * _t) * _unit

        elif _method == EngineMethod.MC.value:
            from utils.monte_carlo import MonteCarlo
            _iteration = _param.get(EngineParam.MCIteration.value)
            if not _iteration:
                raise ValueError("iteration not specified")
            if not isinstance(_iteration, int):
                raise ValueError("type <int> is required for iteration, not {}".format(type(_iteration)))
            _spot = MonteCarlo.stock_price(_iteration, isp=_spot, rate=_rate, div=_div, vol=_vol, t=_t)
            _step = 0.01
            _gamma = [((max(_sign * (_s + 2 * _step - _strike), 0) - max(_sign * (_s - _strike), 0)) -
                       (max(_sign * (_s - _strike), 0) - max(_sign * (_s - 2 * _step - _strike), 0))) /
                       (4 * _step ** 2)
                       for _s in _spot]
```

```
            return average(_gamma) * exp(-_rate * _t) * _unit

        @property
        def type(self):
            """option type - CALL or PUT"""
            if self._type is None:
                raise ValueError("{} type not specified".format(self._name))
            return self._type

        @type.setter
        def type(self, type_):
            if type_ not in option_type:
                raise ValueError("invalid {} type given".format(self._name))
            self._type = type_

        @property
        def strike(self):
            """strike level - percentage of ISP"""
            if self._strike is None:
                raise ValueError("strike level not specified")
            return self._strike

        @strike.setter
        def strike(self, strike_):
            if not isinstance(strike_, float) and not isinstance(strike_, int):
                raise ValueError("type <int> or <float> is required for strike level, not
    {}".format(type(strike_)))
            self._strike = strike_

        @property
        def maturity(self):
            """option maturity - year"""
            if self._maturity is None:
                raise ValueError("maturity not specified")
            return self._maturity

        @maturity.setter
        def maturity(self, maturity_):
            if maturity_ is not None:
                if not isinstance(maturity_, (int, float)):
                    raise ValueError("type <int> or <float> is required for maturity, not
    {}".format(type(maturity_)))
                if maturity_ < 0:
                    raise ValueError("non-negative value is required for maturity, not {}".format(maturity_))
                self._maturity = maturity_

        @staticmethod
        def _load_engine(engine_):
            _method = engine_.get('engine')
            if _method not in [_m.value for _m in EngineMethod]:
                raise ValueError("invalid evaluation engine given: {}".format(_method))
            _param = engine_.get('param', {})
```

```
        return _method, _param

    def _prepare_risk_data(self, mkt_dict_, engine_):
        _load_param = [EnvParam.RiskFreeRate.value, EnvParam.UdSpotForPrice.value,
EnvParam.UdVolatility.value,
                       EnvParam.UdDivYieldRatio.value]
        _rate, _spot, _vol, _div = tuple(self._load_market(mkt_dict_, _load_param))
        _method, _param = self._load_engine(engine_)
        _sign = 1 if self.type == InstType.CallOption.value else -1
        return _rate, _spot, _vol, _div, _method, _param, _sign, self.strike, self.maturity


if __name__ == '__main__':
    import sys

    from personal_utils.logger_utils import get_default_logger
    from personal_utils.time_utils import Timer

    logger = get_default_logger("option pricing test")

    callput = InstType.CallOption.value
    strike = 80
    spot = 100
    maturity = 1
    rate = 2
    vol = 5
    iteration = 1000000

    inst_1 = {
        InstParam.InstType.value: callput,
        InstParam.OptionStrike.value: strike,
        InstParam.OptionMaturity.value: maturity
    }

    mkt = {
        EnvParam.RiskFreeRate.value: rate,
        EnvParam.UdVolatility.value: vol
    }

    engine_1 = dict(engine=EngineMethod.BS.value)
    engine_2 = dict(engine=EngineMethod.MC.value, param={EngineParam.MCIteration.value:
iteration})

    option_1 = Instrument.get_inst(inst_1)

    _timer = Timer("option pricing: {} {}, {} years, rate {}%, vol {}%".format(
        strike, "call" if callput == InstType.CallOption.value else "put", maturity, rate, vol), logger,
rounding_=6)
    price_bs = round(option_1.pv(mkt, engine_1), 6)
    logger.info("price = {} (Black-Scholes)".format(price_bs))
    _timer.mark("pricing using Black-Scholes")
    price_mc = round(option_1.pv(mkt, engine_2), 6)
```

```
    logger.info("price = {} (Monte-Carlo, {} iteration)".format(price_mc, iteration))
    _timer.mark("pricing using Monte-Carlo with {} iteration".format(iteration))
    _timer.close()

    option_1.price = price_bs
    option_1.unit = 1
    logger.info("option payoff at spot {}: {}".format(spot, round(option_1.payoff(spot), 6)))
```

- instrument\portfolio.py

```
# coding=utf-8
"""definition of portfolio for payoff estimation"""

from copy import deepcopy
from enum import Enum
from instrument import InstType, option_type
from instrument.default_param import env_default_param
from instrument.env_param import EnvParam
from numpy import arange, transpose


class CurveType(Enum):
    """supported curve type for portfolio curve generator"""
    Payoff = 'Payoff'
    NetPayoff = 'Net Payoff'
    PnL = 'PnL'
    PV = 'PV'
    Delta = 'Delta'
    Gamma = 'Gamma'


class Portfolio(object):
    """
    portfolio class
    can estimate all components total payoff
    """
    def __init__(self, inst_list_):
        self._components = inst_list_
        self._components_show = []
        self._mkt_data = None
        self._engine = None
        self._center = env_default_param[EnvParam.UdSpotForPrice.value]
        self._maturity = self._check_maturity()
        self._has_stock = self._check_stock()
        self._func_map = {
            CurveType.Payoff.value: ('payoff', False),
            CurveType.NetPayoff.value: ('net_payoff', False),
            CurveType.PnL.value: ('pnl', True),
            CurveType.PV.value: ('pv', True),
            CurveType.Delta.value: ('delta', True),
            CurveType.Gamma.value: ('gamma', True),
        }
```

```
    def gen_curve(self, type_, margin_=20, step_=1, full_=False):
        """generate x (spot / ISP) and y (payoff or) for portfolio payoff curve"""
        _curve_func = [self._comp_sum(type_)]
        _engine = self._func_map[type_][1]
        if full_:
            for _comp in self._components_show:
                _curve_func.append(_comp.__getattribute__(self._func_map[type_][0]))

        _x = self._x_range(margin_, step_)
        _y = []
        for _spot in _x:
            _mkt = deepcopy(self.mkt_data)
            _mkt[EnvParam.UdSpotForPrice.value] = _spot
            _input = (_mkt, self.engine) if _engine else (_mkt, )
            _y.append([_func(*_input) for _func in _curve_func])
        _y = transpose(_y)
        return _x, _y

    def set_show(self, inst_show_):
        """set components that be plotted with portfolio"""
        self._components_show = list(set(inst_show_) - set(self._components))

    def set_mkt(self, mkt_data_):
        """set market data"""
        self.mkt_data = mkt_data_

    def set_engine(self, engine_):
        """set pricing engine"""
        self.engine = engine_

    def maturity(self):
        """return common maturity of portfolio"""
        return self._maturity

    def center(self):
        """return plotting center"""
        return self._center

    def has_stock(self):
        """return if the portfolio contains stock"""
        return self._has_stock

    @property
    def mkt_data(self):
        """market data"""
        if self._mkt_data is None:
            raise ValueError("market data not specified")
        return self._mkt_data

    @mkt_data.setter
    def mkt_data(self, mkt_data_):
        self._mkt_data = mkt_data_
```

```
    @property
    def engine(self):
        """pricing engine"""
        if self._engine is None:
            raise ValueError("pricing engine not specified")
        return self._engine

    @engine.setter
    def engine(self, engine_):
        self._engine = engine_

    def _comp_sum(self, value_type_):
        def _sum_func(*args):
            return sum([_comp.__getattribute__(self._func_map[value_type_][0])(*args) for _comp in
self._components])
        return _sum_func

    def _x_range(self, margin_, step_):
        _strike_list = [_comp.strike for _comp in self._components if _comp.type in option_type]
        _min = min(_strike_list) if _strike_list else self._center
        _max = max(_strike_list) if _strike_list else self._center
        _dist = max([self._center - _min, _max - self._center])
        _x = arange(max(self._center - _dist - margin_, 0), self._center + _dist + margin_ + step_, step_)
        return _x

    def _check_maturity(self):
        _maturity = set([_comp.maturity for _comp in self._components if _comp.type in option_type])
        if len(_maturity) > 1:
            raise ValueError("maturity of all components should be same")
        return _maturity.pop() if len(_maturity) == 1 else 0

    def _check_stock(self):
        return len(list(filter(lambda x: x.type == InstType.Stock.value, self._components))) > 0
```

- instrument\stock.py

```
# coding=utf-8
"""definition of stock for payoff estimation and pricing"""

from instrument import Instrument
from instrument.env_param import EnvParam
# from numpy.ma import exp


class Stock(Instrument):
    """stock class with basic parameters"""
    _name = "stock"

    def __init__(self, inst_dict_):
        super(Stock, self).__init__(inst_dict_)

    def payoff(self, mkt_dict_):
```

```
        """get stock payoff for given spot"""
        _spot = self._load_market(mkt_dict_, [EnvParam.UdSpotForPrice.value])[0]
        return _spot * self.unit

    def pv(self, mkt_dict_, engine_, unit_=None):
        """no pv calc needed for stock"""
        # _div, _t = tuple(self._load_market(mkt_dict_, [EnvParam.UdDivYieldRatio.value,
EnvParam.PortMaturity.value]))
        _unit = unit_ or self.unit
        _spot = self._load_market(mkt_dict_, [EnvParam.UdSpotForPrice.value])[0]
        return _spot * _unit  # * exp(-_div * _t)

    def delta(self, mkt_dict_, engine_, unit_=None):
        """no delta calc needed for stock"""
        _unit = unit_ or self.unit
        return 1 * _unit

    def gamma(self, mkt_dict_, engine_, unit_=None):
        """no gamma calc needed for stock"""
        _unit = unit_ or self.unit
        return 0 * _unit
```

- utils__init__.py

```
# coding=utf-8
"""common utility functions"""

from numpy.ma import log

PRECISION_ZERO = 10 ** -3


def float_int(string_):
    """convert string to int or float according to its real feature"""
    try:
        _number = float(string_)
        return _number if _number % 1 else int(_number)
    except ValueError:
        return None


def to_continuous_rate(rate_):
    """shift discrete rate to continuous rate"""
    return log(1 + rate_)


def parse_kwargs(kwargs_, parse_list_, alternative_=None):
    """parse kwargs with given parse keys"""
    return tuple([kwargs_.get(_key, alternative_) for _key in parse_list_])
```

- Utils\monte_carlo.py

OptionPayOffer

Team Member: Chenxi Xiang, Tianyi Ji, Peiwen Zhang

```python
# coding=utf-8
"""Monte-Carlo engine"""

from numpy.ma import exp, sqrt
from numpy.random import normal as rand_norm
from utils import parse_kwargs


class MonteCarlo(object):
    """Monte Carlo Engine"""

    @classmethod
    def stock_price(cls, iteration_=1, **kwargs):
        """generate stock spot through stochastic process"""
        _rand = rand_norm(0, 1, iteration_)
        _isp, _rate, _div, _vol, _t = parse_kwargs(kwargs, ['isp', 'rate', 'div', 'vol', 't'], 0)
        return _isp * exp((_rate - _div - _vol ** 2 / 2) * _t + _vol * sqrt(_t) * _rand)
```

References

- https://www.investopedia.com/articles/optioninvestor/03/021403.asp
- https://www.investopedia.com/terms/g/gamma.asp
- https://www.investopedia.com/terms/n/net-payoff.asp
- https://www.investopedia.com/terms/v/vanillaoption.asp