Team Member:

Chenxi Xiang 186003830

Peiwen Zhang 174009097

Tianyi Ji 190003115

Optionpayoffer

A tool for option trading

Content

* Application Description
* Target Users and Application Impact
* System Design
  + Modules
  + Data Structure
  + Algorithms
* Performance Analysis
  + Computing Complexity
  + Running time
* Conclusion
* Output
* Appendix
* Reference

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.

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:

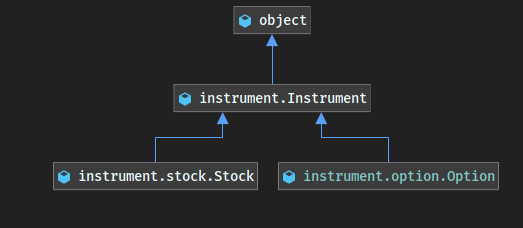
* Industry users: This application can help them save time on writing formulas, changing engines for simulations and realizing visualizations, which will increase their working efficiency.
* College students: This application can help them to deliver a first understanding on option pricing and its dynamics 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.

System Design

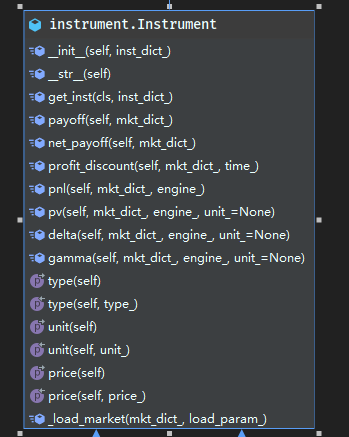
#### Data Structure

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:**

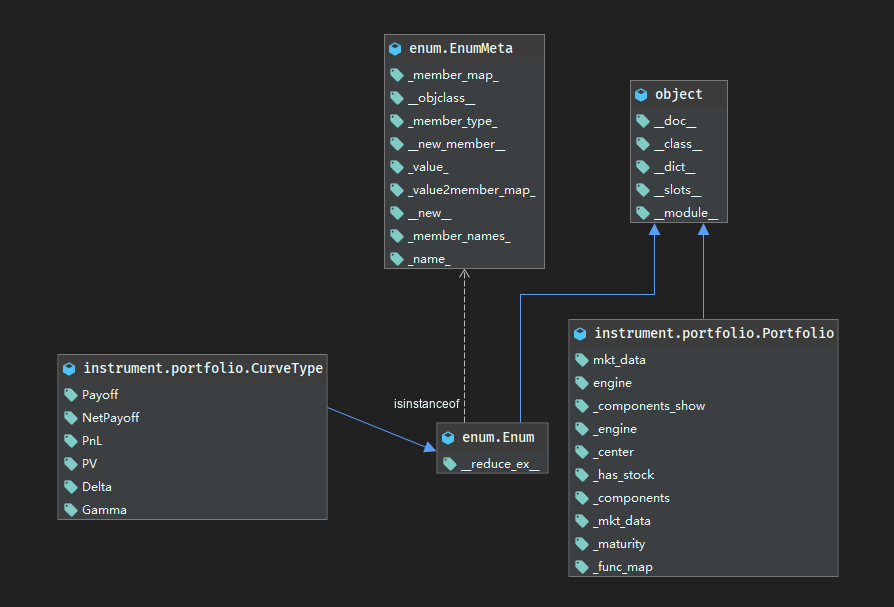


We have a base class named instrument to generate some common properties and methods for option class and stock class.



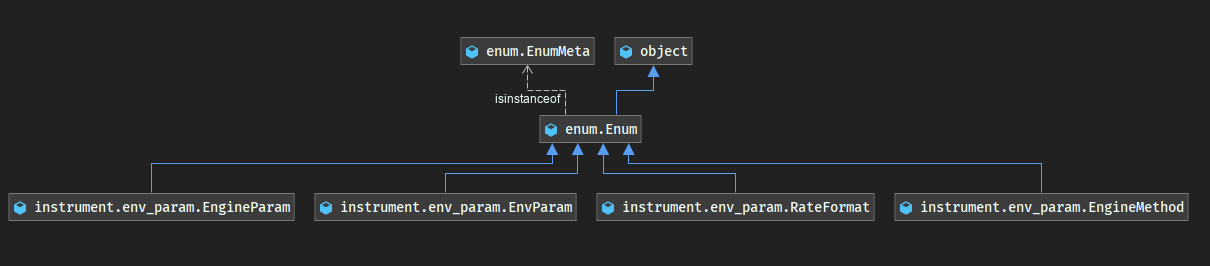
**For Portfolios:**

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.

****

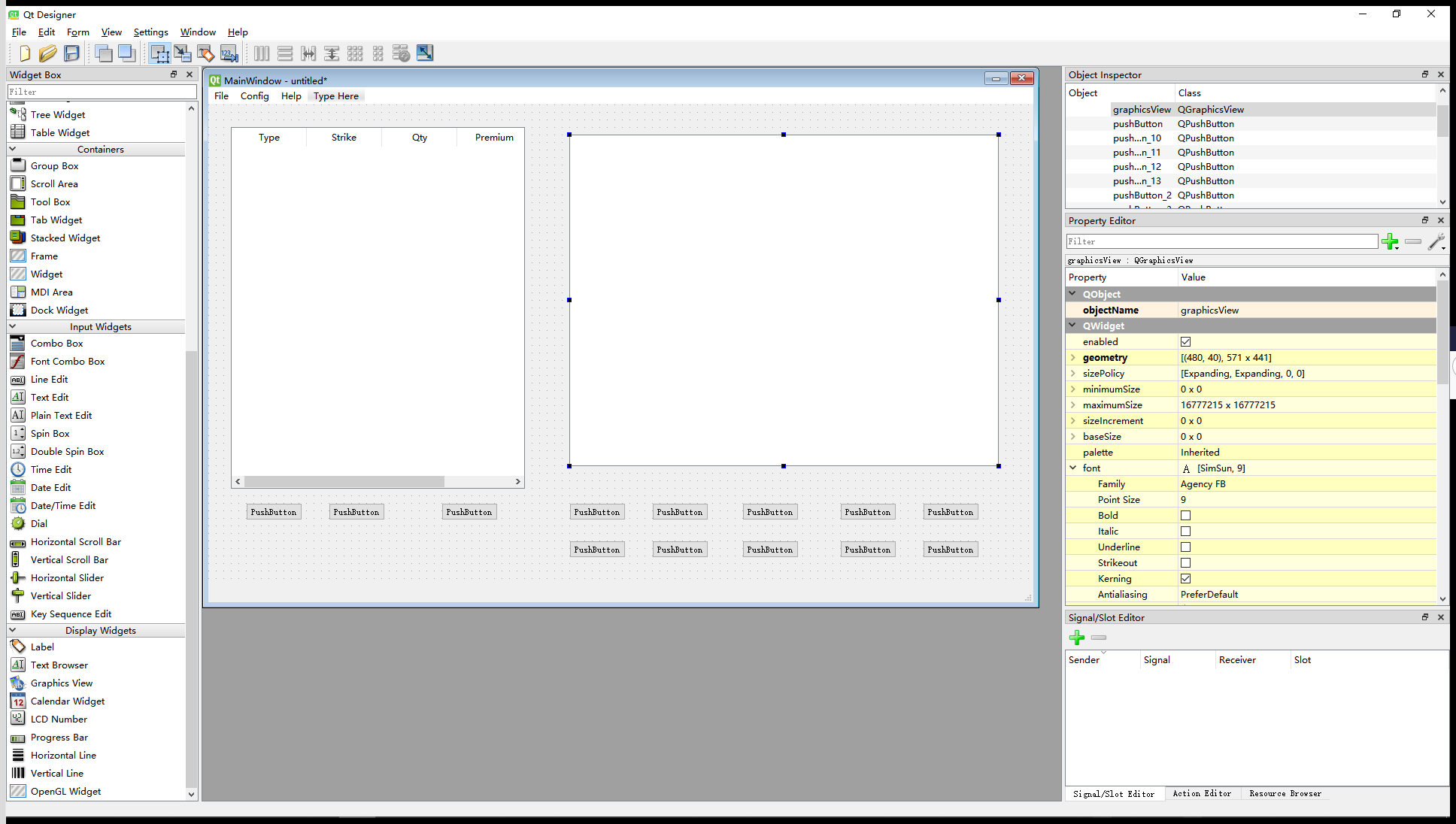
**For Environment Settings:**

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.



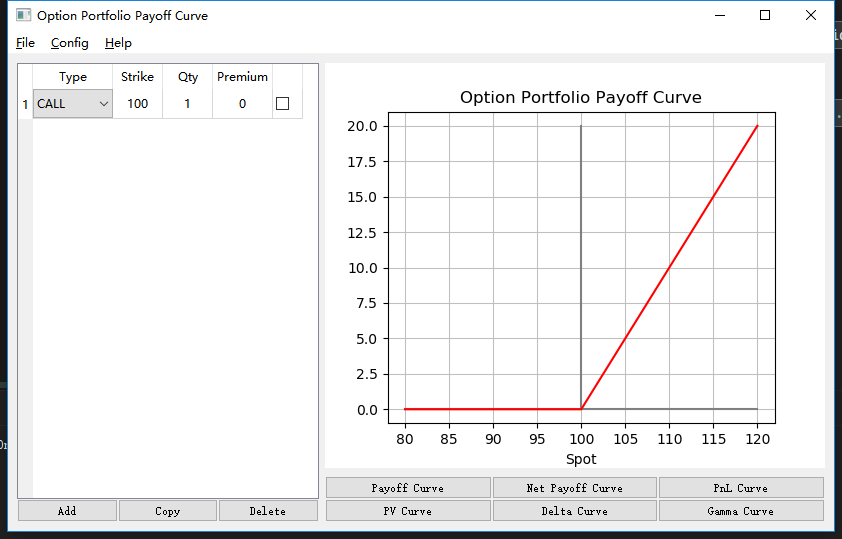
#### GUI Design

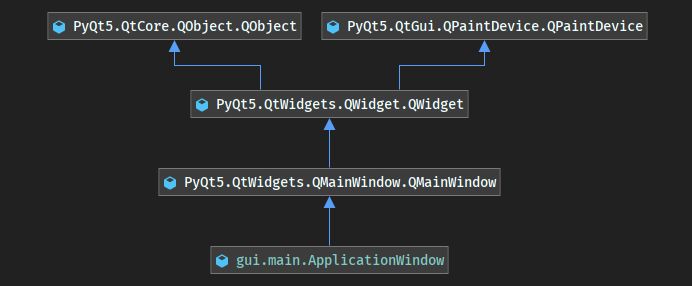
Our project bases on PyQt 5.12.0 with the help of QtDesigner, QtUIC and QtRCC.



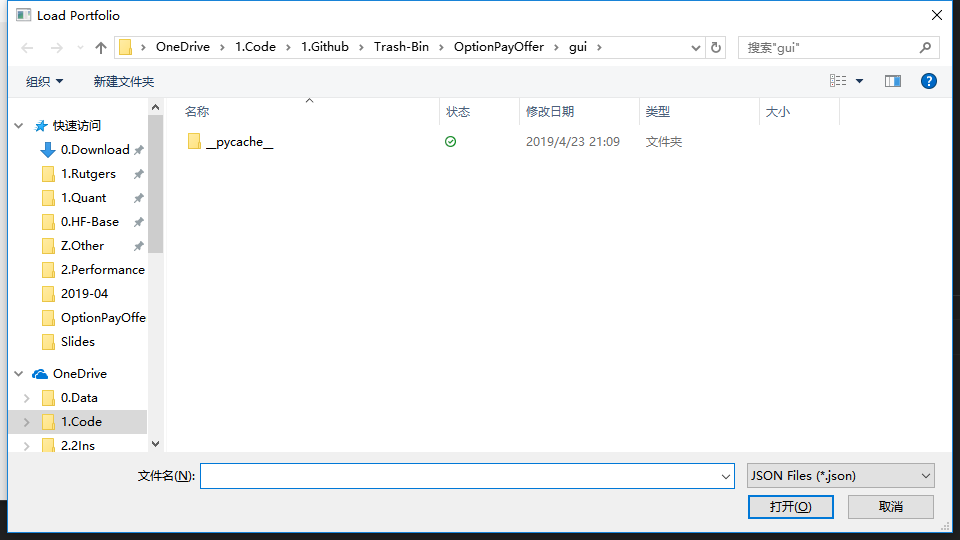
**For MainWindow:**

We get main window by creating an instance of QMainWindow.

****

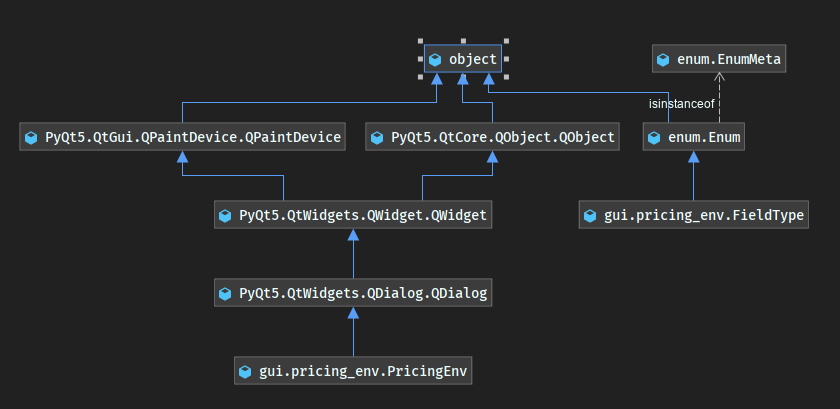
****

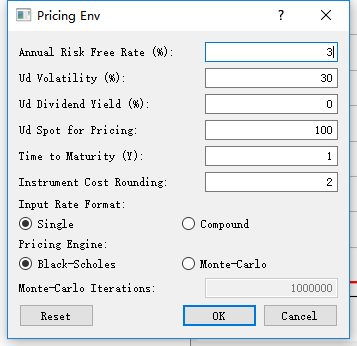
**For Saving / Loading / Experting module:**

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:**

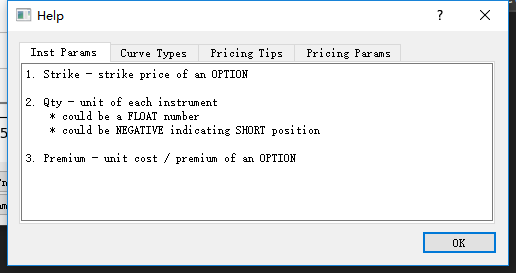
There is a button we created for calling a sub-class of QDialog called PricingEnv. And the structure of PricingEnv shows below:



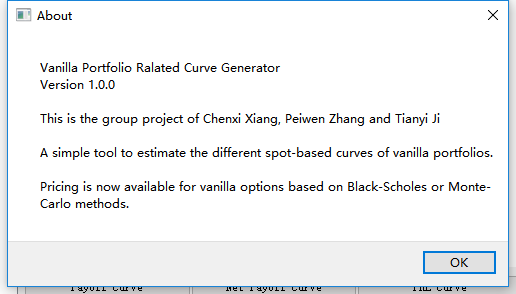


**For more information:**

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

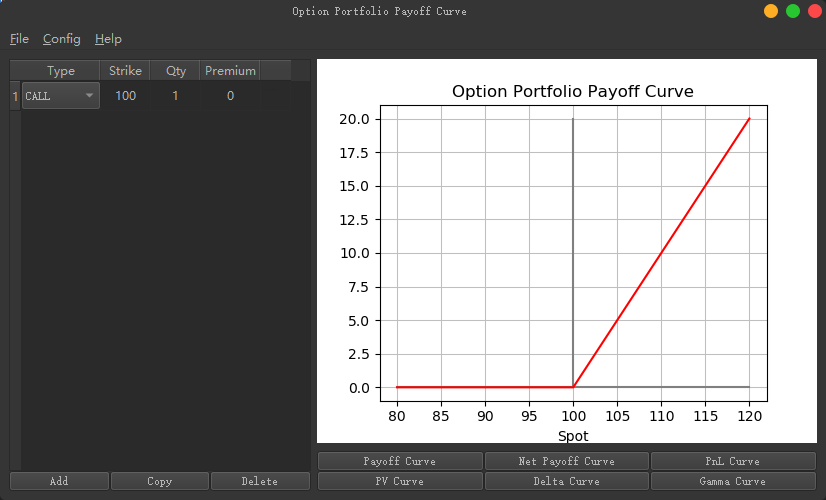
****

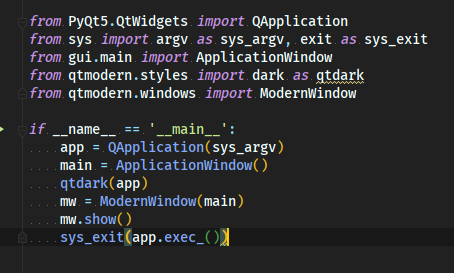
There is also an about dialog to show the description of our program and our teammates.

****

**For Dialogs’ Style:**

We call the qtmodern to make our project more fashion.

****

****

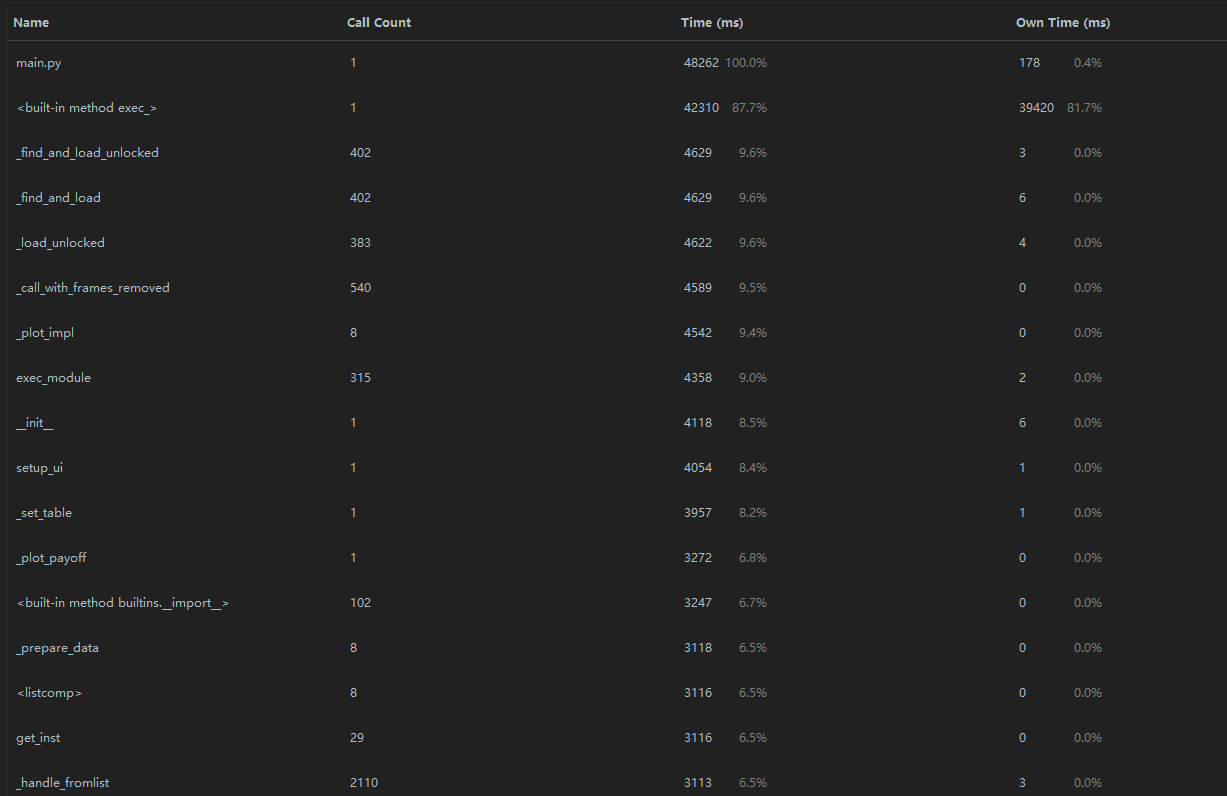
Performance Analysis

#### Computing Complexity

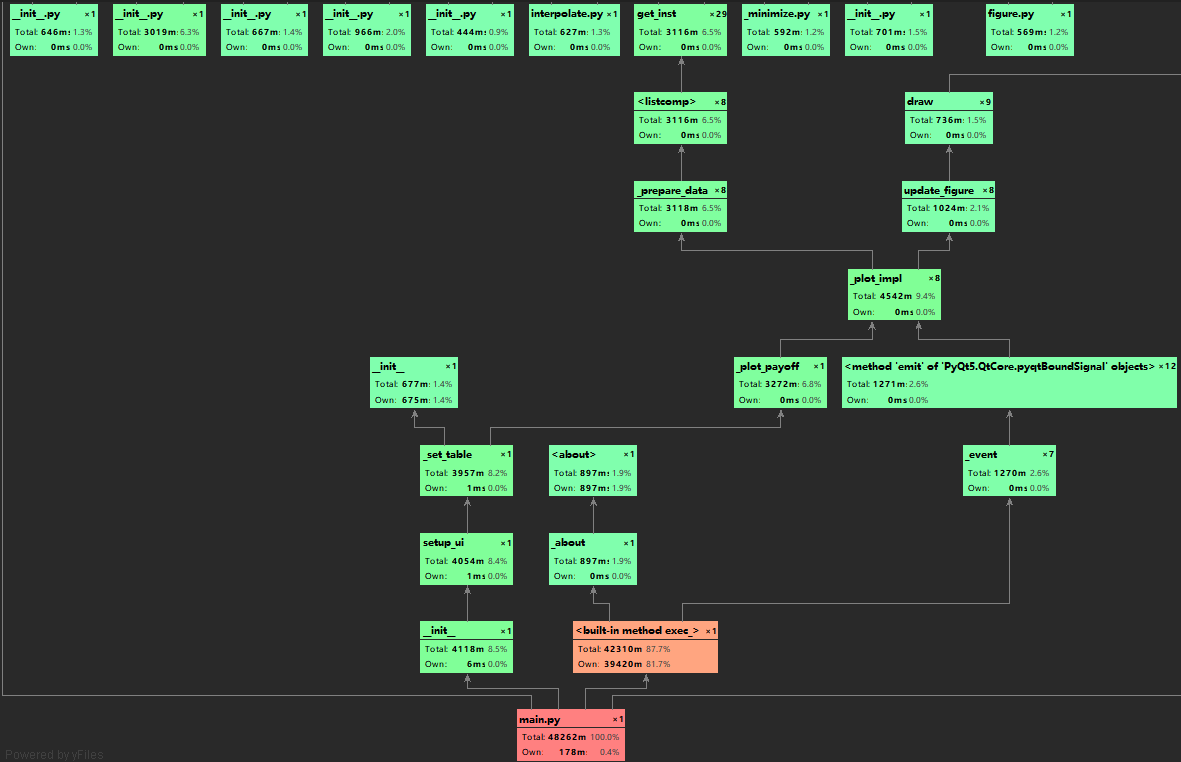
For Calculating each Greeks and numbers like PnL, the module follows their own computing complexity. Generally, Calculating Greeks takes O(n) for each option. And for the overall portfolio, calculating corresponding numbers takes O(N^2).

#### Running Time

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



The Call Graph of OptionPayOffer shows below:



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.

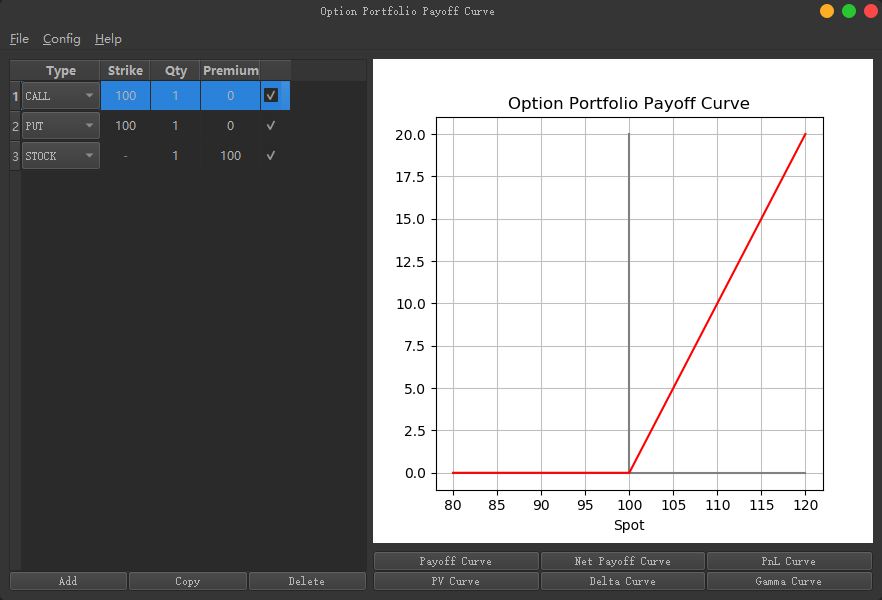
Firstly, we can add the support of real options data with connection to the MySQL database with data in it. It will show the real option Greeks curves to help traders analyze options portfolios better.

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.

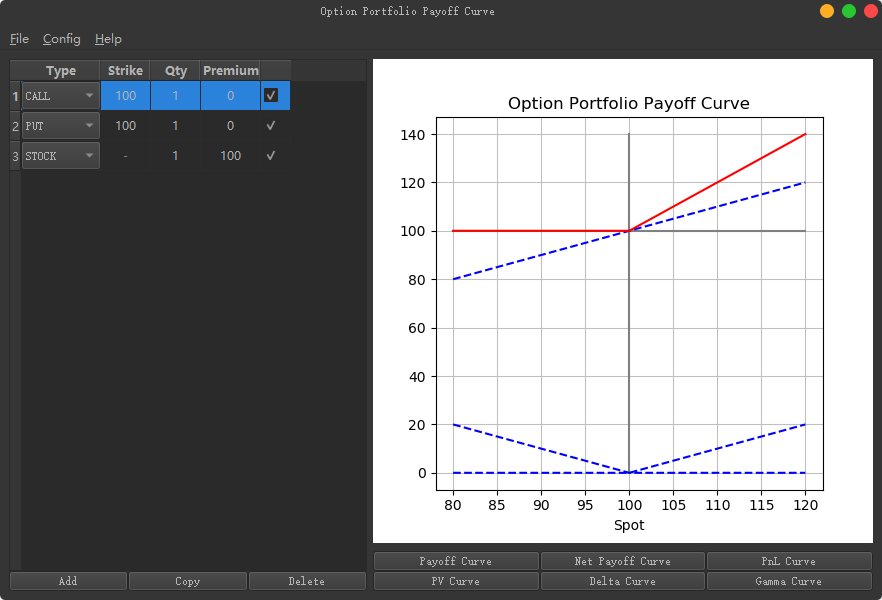
Output

#### A portfolio contains a call, put and stock:

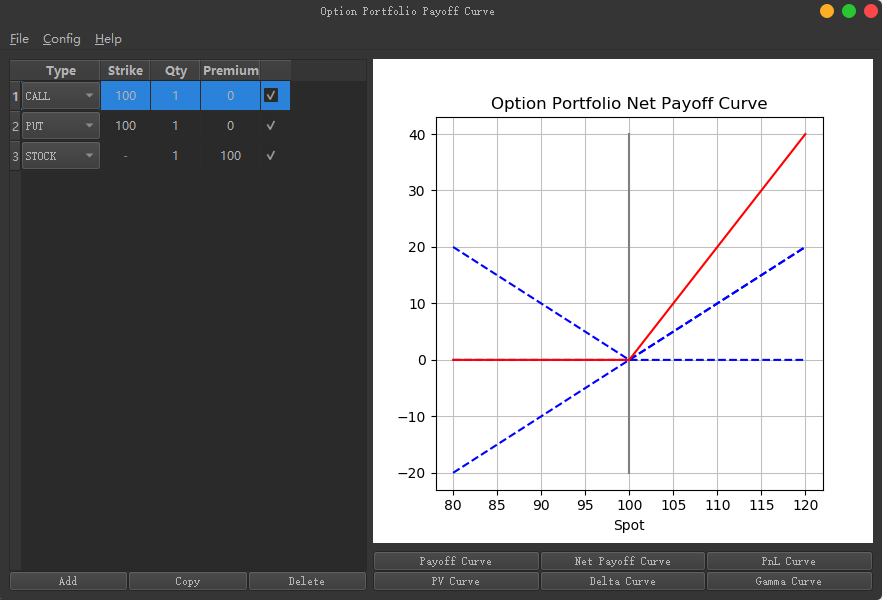


#### Plot some curves we want:

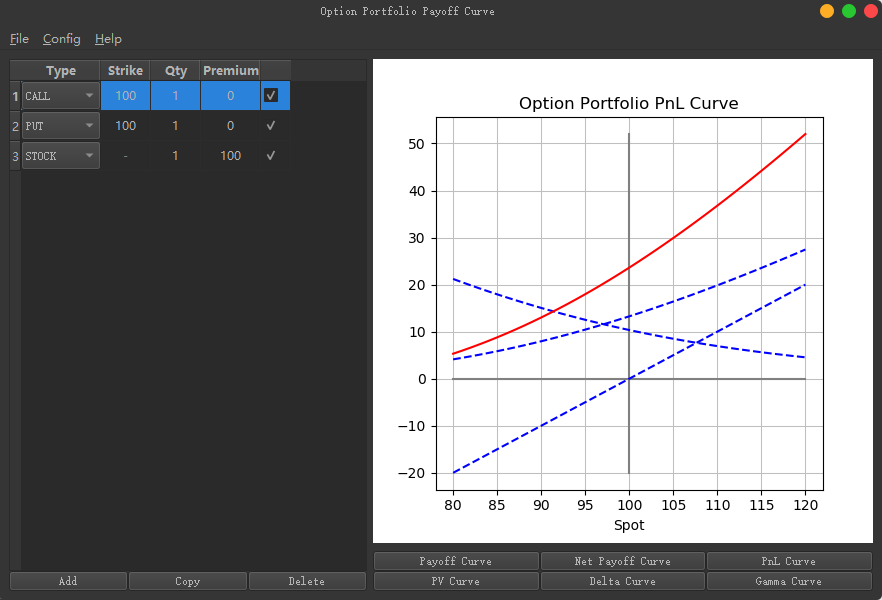
Payoff:



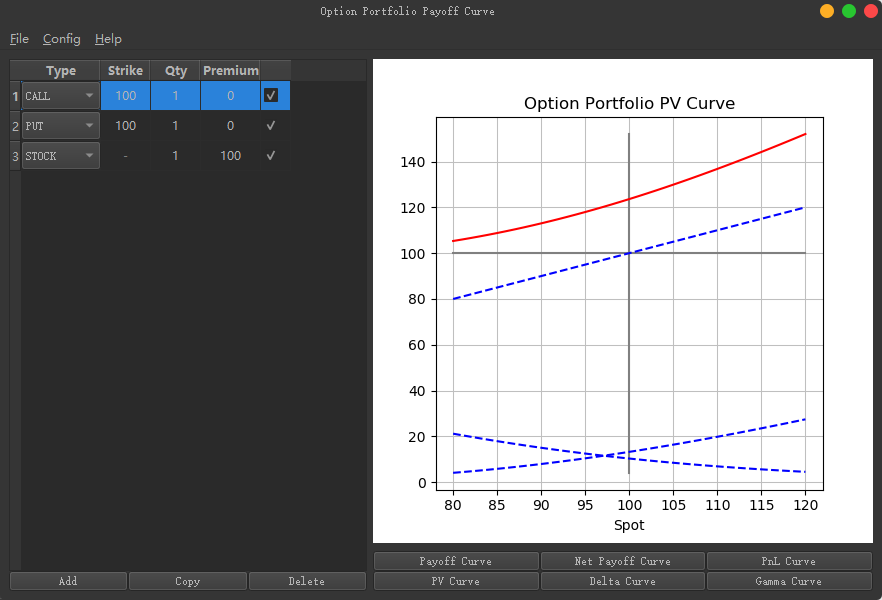
Net Payoff:



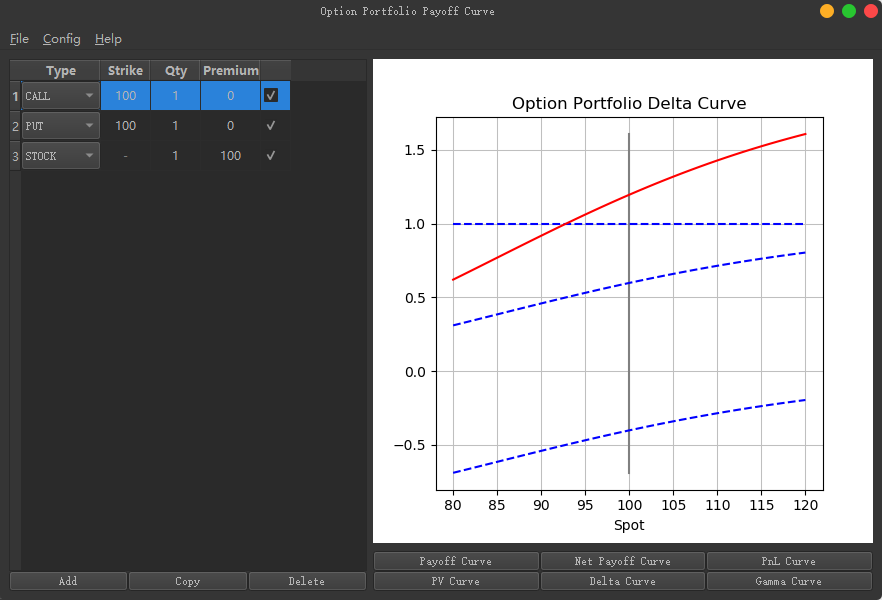
PnL Curve:



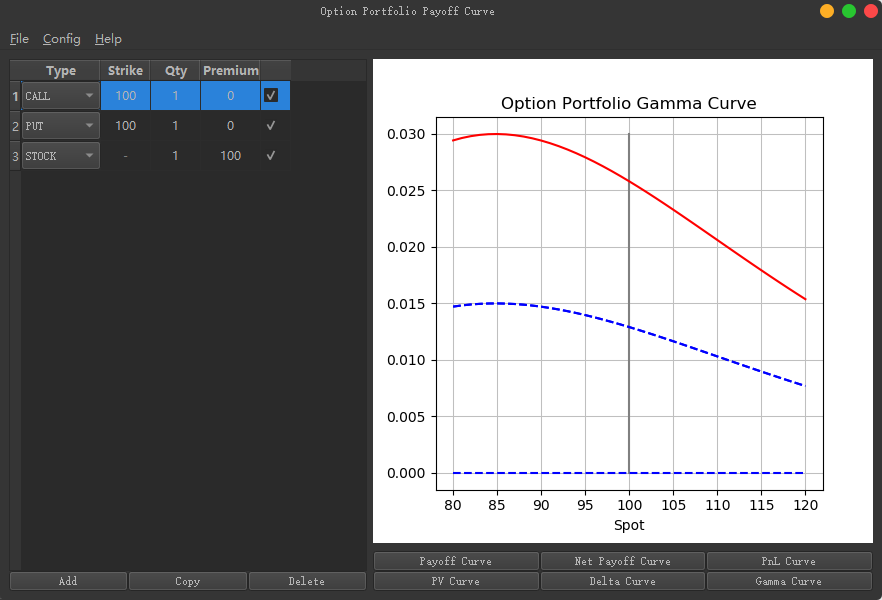
PV Curve:



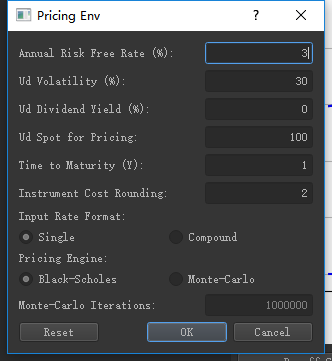
Delta Curve:



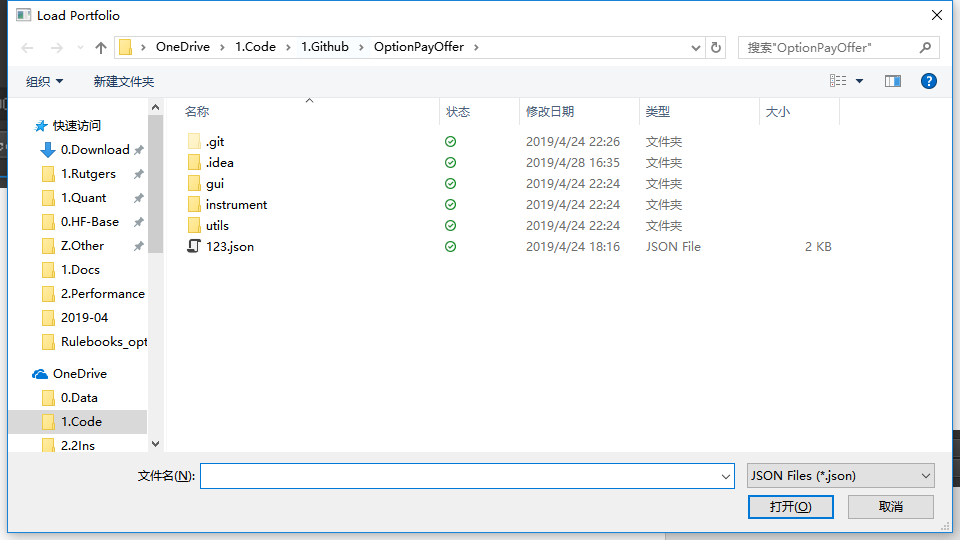
Gamma Curve:



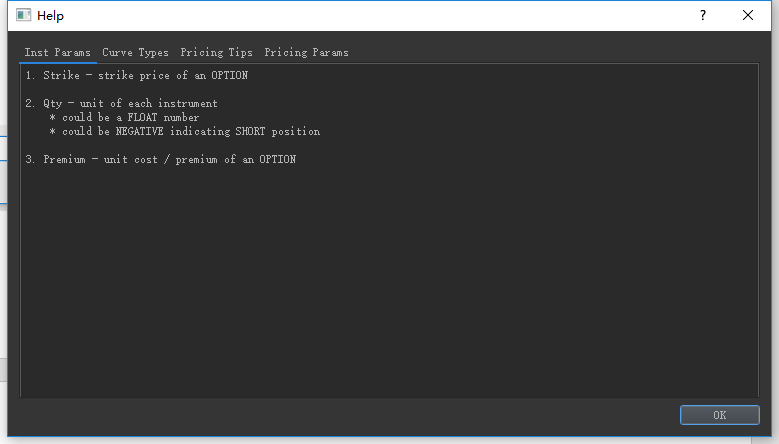
#### Set / Reset pricing environment

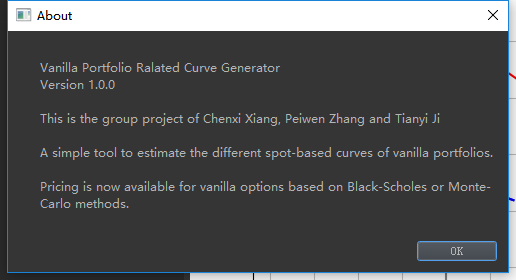


#### Load / Save / Expert Curves



#### Help / About





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
* *# 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
* *# 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