EngFinancialPy

Computatonal Tools for Engineering Financial Decision Making in Python

Kim Leng POH

Department of Industrial Systems Engineering & Management
National University of Singapore
pohkimleng@nus.edu.sg

January 1, 2025

Abstract

This document describes EngFinancialPy which is a set of computational tools implemented in Python for Financial Analysis and Decision Making. The tools are mainly based on the class materials covered in the Engineering Economics and Financial Decision Making courses taught by K.L.Poh at the National University of Singapore. This set of tools in Python supplements the other computational tools in Excel covered in the courses. The source code for the EngFinancialPy module and all the examples may be downloaded from the respective class website.

Contents

1 Introduction

This document describes all the Python Classes and Functions in the EngFinancialPy module. Examples from the Engineering Economics and Financial Decision Making courses are use to illustrate how to use the code.

The souce code for the module and all the examples may be downloaded from the class website in both .py or .jpynb formats. They have been tested on a Python 3.8 environment installed with the Andaconda distribution.

1.1 Getting Started

To use the EngFinancialPy module, simply unzip the souce file and copy EngFinancialPy.py to the same directory or folder where you save your Python .py or Jupyter notebook .ipynb files. No PIP or Conda installlataion is needed.

The packages listed below that are commonly used in numerical computing, data analytics, probability & statistical computing are required. You may have to install them yourself if your Python IDE has not already preinstalled them.

- numpy
- numpy_financial
- matplotlib.pyplot
- pandas
- scipy.stats
- statsmodels.api

1.2 Classes and Functions Dependency

The follow classes and functions are available in EngFinancialPy:

- CF_diagram class
- IntFactor class
- GeomCashFlows class
- Project_CF class
- PnAF function
- pub_Project subclass of Project_CF
- Evaluate_Project function
- OneWaySensit class
- Monte_Carlo_Simulation class
- ATCF_Analysis class
- Asset class
- pprint_list function

Please report any bugs or suggestions for bug fixes to Prof. KL Poh. Suggestions for enhancements to existing code or new features are also welcome.

2 Drawing Cash Flow Diagrams

2.1 Class CF_diagram

[1]:
```
from EngFinancialPy import CF_diagram
```

[2]:
```
print(CF_diagram.__doc__)
```

```
Cash flow diagram plotting Class
   Parameters:
     CashFlows = a complete list or array of cash flows [f0, f1,..., fN], or
                 a sparse unsorted dictionay { time: cash flow value }
     color = color to plot diagram
     currency = currency unit for labels
     time = time unit for x-axis
     figsize = size of figure as(float, float)

   Attributes:
     figure: Figure object
     axes:   Axes object
```

[ ]:
```

```

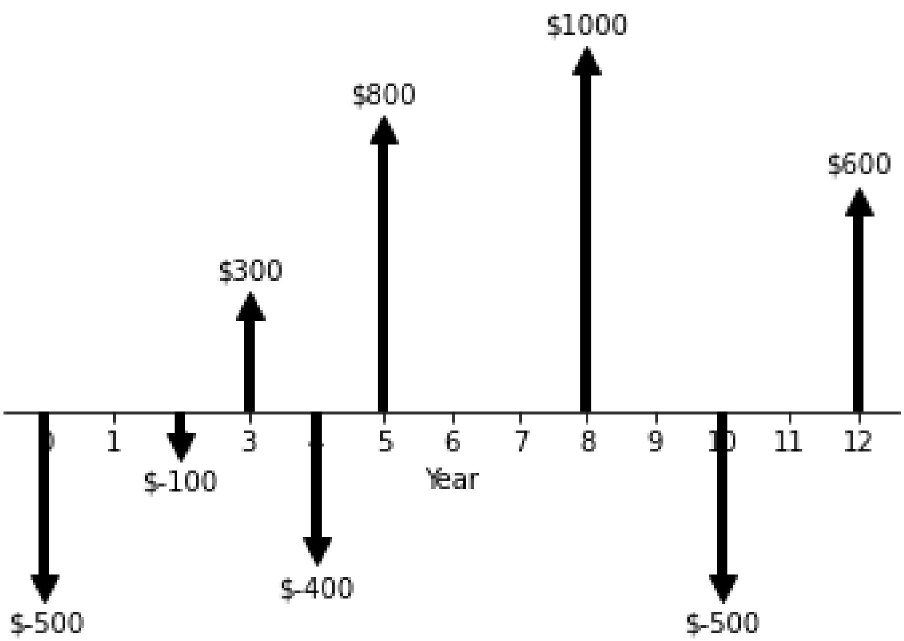
2.2 Examples

Source: 2.8_draw_cash_flows_diagrams.ipynb

[1]:
```
# 2.8_draw_cash_flows_diagrams.ipynb
""" 2.8 Draw Cash Flow Diagrams using CF_diagram class """
import numpy as np
import matplotlib.pyplot as plt
from EngFinancialPy import CF_diagram
```

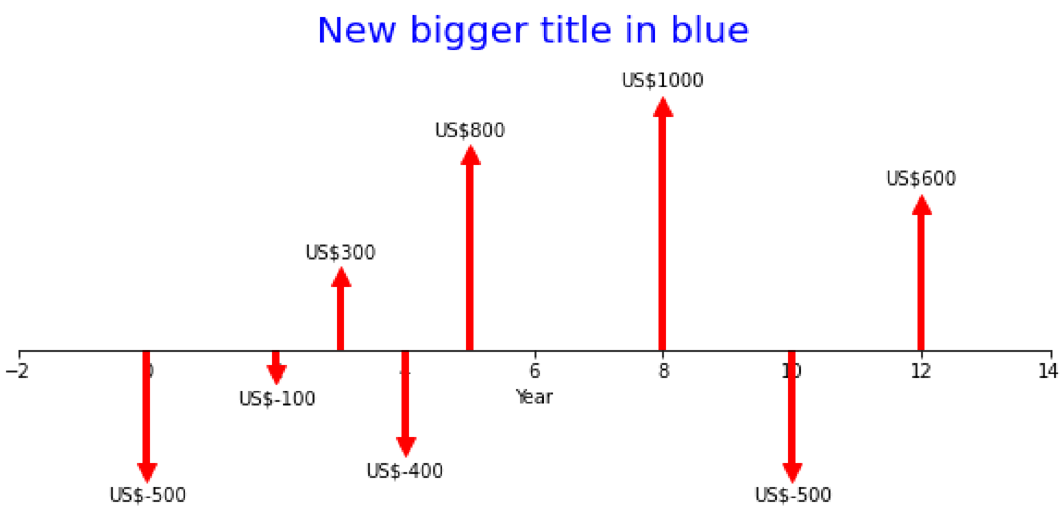
[2]:
```
""" Example 1: Using unsorted dictionary of cash flows as input """
# Cash flows is a dictionary { Time : Cash flow values }
# Time can be in any order, zero cash flow years may be omitted.
CF1_dict = { 0:  -500,
             2:  -100,
             3:   300,
             4:  -400,
             5:   800,
             8:  1000,
             10: -500,
             12:  600 }
```

[3]:
```
 # Create a cash flow diagram object using default parameters
CF_diagram(CF1_dict)
plt.show()
```



[4]:

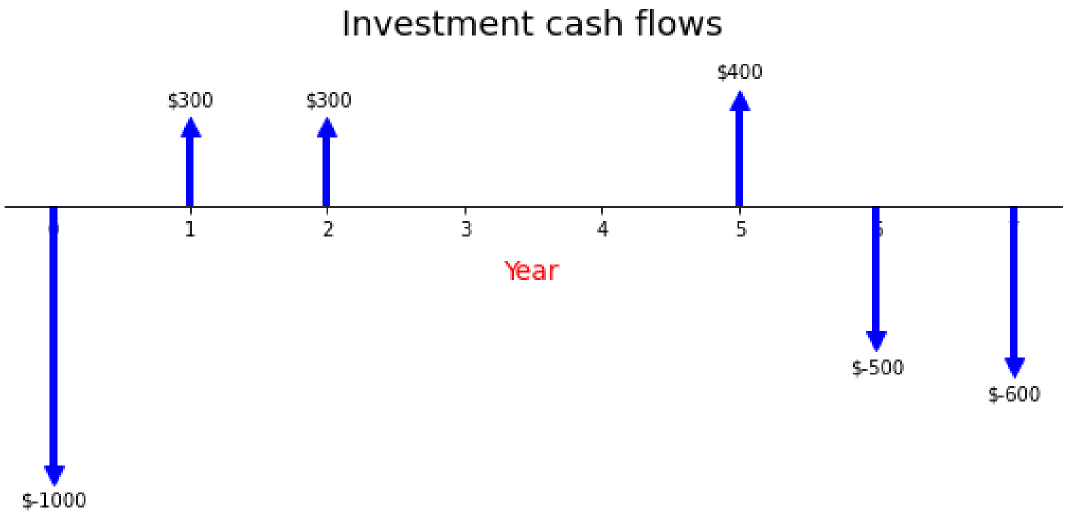
```
# Don't like the above diagram? We can customize it as you like.
D2=CF_diagram(CF1_dict, color='red', currency="US$", time_unit='Year',
              time_start=-2, time_step=2, time_end=15,
              title = "Investment ABC cash flows", figsize=(10,4))

# We can make some minor changes to the instance axes attribute
D2.axes.set_title('New bigger title in blue', fontsize=20, color='blue')
plt.show()
```



[5]:

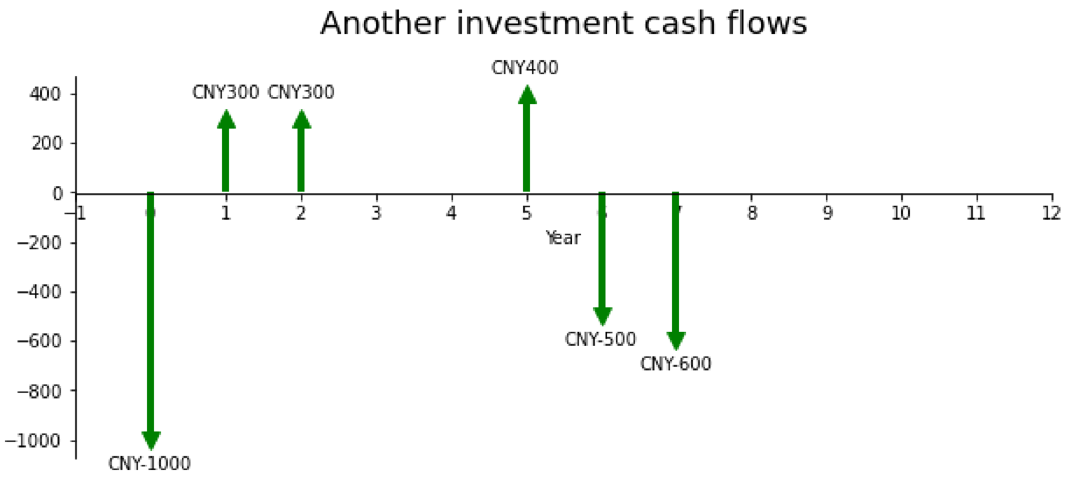
```
""" Example 2: Use List of cash flows in chrological order """
```

```python
# Cash flows is a List enumerating year-by-year cash flows
CF3_list = [-1000, 300, 300, 0, 0, 400, -500, -600]

# Create a cash flow diagram
D3 = CF_diagram(CF3_list, color='blue', figsize=(10,4),
                title="Investment cash flows")

# Change the fontsize of the xlabel and padding from the axis.
D3.axes.set_xlabel("Year", fontsize=14, color='red', labelpad=10)
plt.show()
```



[6]:

```python
""" Example 3: Using np.array of cash flows """

# Cash flows is a year-by-year np.array of cash flow
CF4 = np.array([-1000, 300, 300, 0, 0, 400, -500, -600])

# Create a cash flow diagram
D4 = CF_diagram(CF4, color='green', currency='CNY',
                time_start=-1, time_end=12, figsize=(10, 4),
                title="Another investment cash flows")

# Turn on the y-axis at time_start
D4.axes.get_yaxis().set_visible(True)
D4.axes.spines['left'].set_visible(True)
plt.show()
```




```
[ ]:
```

3 Annuity and Interest Factors

3.1 Class IntFactor

```
[1]: from EngFinancialPy import IntFactor
```

```
[2]: print(IntFactor.__doc__)
```

```
 A Class for Interest Factors for discrete cash flows with
        discrete and continuous compounding.
        Parameters:
          find = 'P', 'F' or 'A'
          given = 'P', 'F', A' or 'G'
          rate = interest rate
          nper = number of periods
          continuous = optional, set to True if continuous compounding
        Attributes:
          value = value of the interest factor
          symbol = string [X/Y, rate, nper ]
          params = dictionary of all the interest factor's parameters
          compounding = string 'Discrete' or 'Continuous'
```

```
[ ]:
```

3.2 Examples (2.8)

Source: 2.8_compute_interest_factors.ipynb

```
[1]: # 2.8_compute_interest_factors.ipynb
     """ 2.8 Compute interest factors for discrete case flows under
         discrete and continuous compounding using IntFactor class """
```

```
from EngFinancialPy import IntFactor
```

```
[2]: """" Example 1: Compute the value of [P/G, 8%, 10] """

     # Method 1: Compute a factor's value directly and print it.
     print(IntFactor('P', 'G', 0.08, 10).value)
```

```
25.976831476182575
```

```
[3]: # Method 2: Create the interest factor and then use its methods.
     factor = IntFactor('P', 'G', 0.08, 10)
     # print out the value
     print(f"{factor.value:.6f}")
     # Print out the symbol notation
     print(factor.symbol)
     # See what are the factor's parameters
     print(factor.params)
```

```
25.976831
[P/G, 0.08, 10]
{'find': 'P', 'given': 'G', 'rate': 0.08, 'nper': 10, 'continuous': False}
```

```
[4]: """ Example 2: Printing all the interest factor values for 12%, 10 periods
                    under discrete & continuous compoudings """
     # List of all the find and given parameters to compute
     FG = ('FP','PF','FA','AF','PA','AP','PG','FG','AG')
     rate = 0.12
     Nper = 10
```

```
[5]: print("\nDiscrete Compounding:")
     for fg in FG:
         factor = IntFactor(*fg, rate, Nper)
         print(f"  {factor.symbol} = {factor.value:12.8f}")

     print("\nContinuous Compounding:")
     for fg in FG:
         factor = IntFactor(*fg, rate, Nper, continuous=True)
         print(f"  {factor.symbol} = {factor.value:12.8f}")
```

```
Discrete Compounding:
  [F/P, 0.12, 10] =   3.10584821
  [P/F, 0.12, 10] =   0.32197324
  [F/A, 0.12, 10] =  17.54873507
  [A/F, 0.12, 10] =   0.05698416
  [P/A, 0.12, 10] =   5.65022303
  [A/P, 0.12, 10] =   0.17698416
  [P/G, 0.12, 10] =  20.25408885
  [F/G, 0.12, 10] =  62.90612558
  [A/G, 0.12, 10] =   3.58465299

Continuous Compounding:
  [F/P, r=0.12, 10] =   3.32011692
```

```
[P/F, r=0.12, 10] =   0.30119421
[F/A, r=0.12, 10] =  18.19744483
[A/F, r=0.12, 10] =   0.05495277
[P/A, r=0.12, 10] =   5.48096505
[A/P, r=0.12, 10] =   0.18244962
[P/G, r=0.12, 10] =  19.36536397
[F/G, r=0.12, 10] =  64.29527262
[A/G, r=0.12, 10] =   3.53320333
```

[ ]:

3.3 Examples on equivalent values of discrete cash flows (2.2)

Source: 2.2_equivalent_values_of_discrete_cash_flows_examples.ipynb

[1]:
```
# 2.2_equivalent_values_of_discrete_cash_flows_examples.ipynb
""" 2.2 Equivalent Values of Discrete Cash Flows """
from EngFinancialPy import IntFactor
import numpy_financial as npf
```

[2]:
```
""" Example 1
    Suppose you invest $8,000 in a saving account that earns 10%
    compound interest per year.  What is the amount in the account
    at the end of 4 years?
"""
# Using IntFactor class
F = 8000 * IntFactor('F','P', 0.1, 4).value
print(f"Amount at the end of 4 years = ${F:,.2f}")
```

```
Amount at the end of 4 years = $11,712.80
```

[3]:
```
# Using npf.fv function directly
F = - npf.fv(0.1, 4, 0, 8000)
print(f"Amount at the end of 4 years = ${F:,.2f}")
```

```
Amount at the end of 4 years = $11,712.80
```

[4]:
```
""" Example 2
    An investment is to be worth $10,000 in six years.
    If the return on investment 8% per year compounded yearly,
    how much should be invested today?
"""
# Using IntFactor class
P = 10000 * IntFactor('P', 'F', 0.08, 6).value
print(f"Amount to be invested now = ${P:,.2f}")
```

```
Amount to be invested now = $6,301.70
```

[5]:
```
# Using npf.pv function directly
P = -npf.pv(0.08, 6, 0, 10000)
print(f"Amount to be invested now = ${P:,.2f}")
```

```
Amount to be invested now = $6,301.70
```

[6]:
```
""" Example 3
    15 equal deposits of $1,000 each will be made into a bank account
    paying 5% compound interest per year, the first deposit being one
    year from now.  What is the balance exactly 15 years from now?
"""
# Using IntFactor class
F = 1000 * IntFactor('F', 'A', 0.05, 15).value
print(f"Balance at EoY 15 = ${F:,.2f}")
```

```
Balance at EoY 15 = $21,578.56
```

[7]:
```
# Using npf.fv function directly
F = -npf.fv(0.05, 15, 1000, 0)
print(f"Balance at EoY 15 = ${F:,.2f}")
```

```
Balance at EoY 15 = $21,578.56
```

[8]:
```
""" Example 4
    What is the equivalent present value of a series of end-of-year
    equal incomes valued at $20,000 each for 5 years if the interest
    rate is 15% per year?
"""
# Using IntFactor class
P = 20_000 * IntFactor('P', 'A', 0.15, 5).value
print(f"Equivalent present vlaue = ${P:,.2f}")
```

```
Equivalent present vlaue = $67,043.10
```

[9]:
```
# Using npf.pv function directly
P = -npf.pv(0.15, 5, 20_000, 0)
print(f"Equivalent present vlaue = ${P:,.2f}")
```

```
Equivalent present vlaue = $67,043.10
```

[10]:
```
""" Example 5
    If you need a lump sum of $1 million at your retirement 45 years
    from now, how much must you save per year if the interest rate
    is 7% per year?
"""
# Using IntFactor class
A = 1_000_000 * IntFactor('A', 'F', 0.07, 45).value
print(f"Saving per year = ${A:,.2f}")
```

```
Saving per year = $3,499.57
```

[11]:
```
# Using npf.pmt function directly
A = - npf.pmt(0.07, 45, 0, 1_000_000)
print(f"Saving per year = ${A:,.2f}")
```

```
Saving per year = $3,499.57
```

[12]:
```
""" Example 6
    Consider a loan of $8,000 to be paid back with 4 equal EoY
    installments? What is the yearly repayment amount if the
```

```
    interest rate is 10%?
"""
# Using IntFactor class
A = 8_000 * IntFactor('A', 'P', 0.1, 4).value
print(f"EoY payment amount = ${A:,.2f}")
```

```
EoY payment amount = $2,523.77
```

[13]:
```
# Using npf.pmt function directly
A = - npf.pmt(0.1, 4, 8_000, 0)
print(f"EoY payment amount = ${A:,.2f}")
```

```
EoY payment amount = $2,523.77
```

[14]:
```
""" Example 7
    You intend to rent a room for 12 months during your overseas
    exchange program. The landlord asks for a monthly rent of $1,000,
    payable at the beginning of each month.
"""

# (a) If you wish to pay the rents at the end of each month instead,
#     what amount should you pay if the time value of money to the
#     landlord is 2% per month?

B = 1_000

# Using IntFactor class
A = B * IntFactor('F','P', 0.02, 1).value
print(f"End-of-month rent = ${A:,.2f}")
```

```
End-of-month rent = $1,020.00
```

[15]:
```
# Using npf.fv function directly
A = - npf.fv(0.02, 1, 0, B)
print(f"End-of-month rent = ${A:,.2f}")
```

```
End-of-month rent = $1,020.00
```

[16]:
```
# (b) If you wish to pay all the rents with one lump sum upon moving in
#     instead, what amount should you pay if the time value of money to
#     the landlord is 2% per month?

# Using IntFactor class
P = B * (1 + IntFactor('P','A', 0.02, 11).value)
print(f"One lump sum payment now = ${P:,.2f}")
```

```
One lump sum payment now = $10,786.85
```

[17]:
```
# Using npf.pv function directly
P = - npf.pv(0.02, 12, B, 0, when=1)
print(f"One lump sum payment now = ${P:,.2f}")
```

```
One lump sum payment now = $10,786.85
```

[18]:
```python
# (c) If you wish to pay all the rents with one lump on moving out
#     at the end of 12 months, what amount should you pay if the time
#     value of money to the landlord is 2% per month?

# Using IntFactor class
F = B * IntFactor('F','A',0.02,12).value*IntFactor('F','P',0.02,1).value
print(f"One lump sum payment at EoY 12 = ${F:,.2f}")
```

```
One lump sum payment at EoY 12 = $13,680.33
```

[19]:
```python
# Using npf.fv function directly
F = - npf.fv(0.02, 12, B, 0, when=1)
print(f"One lump sum payment at EoY 12 = ${F:,.2f}")
```

```
One lump sum payment at EoY 12 = $13,680.33
```

[20]:
```python
""" Example 8
    Given the following cash flows
      Year:      0    1    2    3    4    5    6    7    8
      Amount:    0  100  106  112  118  124  130  136  142
"""

# (a)  Find the equivalent present value at 10%
# Using IntFactor class
P = 100 * IntFactor('P','A', 0.1, 8).value +  \
      6 * IntFactor('P','G', 0.1, 8).value
print(f"Equivalent present vlaue = ${P:,.2f}")
```

```
Equivalent present vlaue = $629.66
```

[21]:
```python
# Using npf.npv function directly
P = npf.npv(0.1, [0, 100, 106, 112, 118, 124, 130, 136, 142])
print(f"Equivalent present value = ${P:,.2f}")
```

```
Equivalent present value = $629.66
```

[22]:
```python
# (b) Find the equivalent annual value at 10%

# Using IntFactor class
A = 100 + 6 * IntFactor('A','G', 0.1, 8).value
print(f"Equivalent annual value = ${A:,.2f}")
```

```
Equivalent annual value = $118.03
```

[23]:
```python
# Using npf.npv and npf.pmt functions directly
A = - npf.pmt(0.1, 8,
      npf.npv(0.1, [0, 100, 106, 112, 118, 124, 130, 136, 142]), 0)
print(f"Equivalent annual value = ${A:,.2f}")
```

```
Equivalent annual value = $118.03
```

[24]:
```python
# (c) Find the equivalent future value at 10%

# Using IntFactor class
```

```
F = 100 * IntFactor('F','A', 0.1, 8).value +  \
      6 * IntFactor('F','G', 0.1, 8).value
print(f"Equivalent future value = ${F:,.2f}")
```

```
Equivalent future value = $1,349.74
```

[25]:
```
# Using npf.npv and npf.fv functions directly
F = - npf.fv(0.1, 8, 0,
      npf.npv(0.1, [0, 100, 106, 112, 118, 124, 130, 136, 142]))
print(f"Equivalent future value = ${F:,.2f}")
```

```
Equivalent future value = $1,349.74
```

3.4 Examples on equivalent values of discrete cash flows continuous compounding (2.5)

Source: 2.5_continuous_compounding_of_discrete_CF.ipynb

[1]:
```
# 2.5_continuous_compounding_of_discrete_CF.ipynb
"""  2.5 Continuous compounding of discrete cash flows """
from EngFinancialPy import IntFactor
import numpy as np
import numpy_financial as npf
```

[2]:
```
""" Example 1 (2.5)
    Consider a loan of $1,000.  What equivalent uniform end-of-year
    payments must be made for 10 years if the nominal interest rate
    is 10% per year compound continuously?
"""
P = 1000
r1 = 0.1   # per year
N1 = 10    # years

# Using IntFactor class
A1 = P * IntFactor('A','P',  r1, N1, continuous=True).value
print(f"Uniform EoY payment amount = {A1:,.2f}")
```

```
Uniform EoY payment amount = 166.38
```

[3]:
```
# Using npf.pmt function directly
A1 = - npf.pmt(np.exp(r1)-1, N1, P, 0)
print(f"Uniform EoY payment amount = {A1:,.2f}")
```

```
Uniform EoY payment amount = 166.38
```

[4]:
```
""" Example 2 (2.5)
    In the previous example, what is the repayment amount
    if it is to be made at the end of every six months instead?
"""
r2= r1/2    # per six months
N2 = 2*N1   # six-month periods

# Using IntFactor class
```

```python
A2 = P * IntFactor('A','P',  r2, N2, continuous=True).value
print(f"Uniform semi-annual payment amount = {A2:,.2f}")
```

```
Uniform semi-annual payment amount = 81.11
```

[5]:
```python
# Using npf.pmt function directly
A2 = - npf.pmt(np.exp(r2)-1, N2, P, 0)
print(f"Uniform semi-annual payment amount = {A2:,.2f}")
```

```
Uniform semi-annual payment amount = 81.11
```

[ ]:
```python

```

4 Geometric Series Cash Flows Analysis

4.1 Class GeomCashFlows

[1]:
```python
from EngFinancialPy import GeomCashFlows
```

[2]:
```python
print(GeomCashFlows.__doc__)
```

```
 A Class for Geometric Series Cash Flows
        Parameters:
          rate = effective interest rate
          nper = number of periods
          A1   = cash flow at end of year 1
          growth = year-on-year growth rate of annual flows
        Attributes:
          P = Present equivalent value of cash flows
          A = Equivalent uniform annual cash flows
          F = Future equivalent value of cash flows
          G = Equivalent uniform gradient cash flows (0,0,G,2G,...,(n-1)G)
          params = dictionary of cash flow parameters
```

[ ]:
```python

```

4.2 Examples (2.8)

Source: 2.8_geometric_cash_flows_analysis.ipynb

[1]:
```python
# 2.8_geometric_cash_flows_analysis.ipynb
""" 2.8 Geometric Cash Flow series analsyis using GeomCashFlows class """
from EngFinancialPy import GeomCashFlows
```

[2]:
```python
# Geometric cash flows example in Section 2.2.9
i  = 0.25
f  = 0.20
A1 = 1000
N = 4

# Create a Geometric cash flow series
```

```
GCF = GeomCashFlows(i, N, A1, f)
```

[3]:
```
# Determine its equivalent P, A, F and G values
print(f"P = {GCF.P:,.2f}")
print(f"A = {GCF.A:,.2f}")
print(f"F = {GCF.F:,.2f}")
print(f"G = {GCF.G:,.2f}")
```

```
P = 3,013.07
A = 1,275.86
F = 7,356.13
G = 1,041.58
```

[4]:
```
# Get the parameters
print(GCF.params)
```

```
{'rate': 0.25, 'nper': 4, 'A1': 1000, 'growth': 0.2}
```

[ ]:
```

```

4.3 Examples on computing equivalent values of Geometric series cash flows

Source: 2.2.9_equivalent_values_of_Geometric_series_CF.ipynb

[1]:
```
# 2.2.9_equivalent_values_of_Geometric_series_CF.ipynb
"""  2.2.9 Equivalent values of Geometric series cash flows """
from EngFinancialPy import GeomCashFlows, IntFactor
```

[2]:
```
""" Example (2.2.9)
Find the equivalent present value of the following cash flows
if the interest rate is 25% per year:
    EoY   0  1     2          3            4
    CF    0  1000  1000(1.2)  1000(1.2)^2  1000(1.2)^3
"""
# Parameters
i  = 0.25
f  = 0.20
A1 = 1000
N = 4
```

[3]:
```
# Using GeomCashFlows class
geomCF = GeomCashFlows(i, N, A1, f)
P = geomCF.P
print(f"Equivalent PV = {P:,.2f}")
```

```
Equivalent PV = 3,013.07
```

[4]:
```
# Using interest factor formulas
P = A1*(1 - IntFactor('P','F',i,N).value    \
          * IntFactor('F','P',f,N).value)   \
       / (i - f)
print(f"Equivalent PV = {P:,.2f}")
```

```
Equivalent PV = 3,013.07
```

```
[ ]:
```

5 Financial Analysis of Single Projects

5.1 Class Project_CF

```
[1]: from EngFinancialPy import Project_CF
```

```
[2]: print(Project_CF.__doc__)
```

```
 Project Cash Flows Class for profitability, liquality and
        feasibility analysis.
    Parameters:
      cash_flows = Array of cash flows starting from time 0.
                   If not defined, must be set later by set_cf method.
      marr = Project MARR. If undefined, must be either set with
              set_marr method or given when computing profitability measures.
    Attributes:
      cf = Project cash flows series
      life = Project life
      marr = Project MARR
      name = Project name
    Methods:
      set_marr(marr): Set the project Marr
      set_cf(CF): Set the project cash flows
      pw(marr): Compute PW at marr. Project MARR is used if marr is not given.
      npv(marr): Compute PW at marr. Project MARR is used if marr is not given.
      aw(marr): Compute AW at marr. Project MARR is used if marr is not given.
      fw(marr): Compute FW at marr. Project MARR is used if marr is not given.
      irr : Compute project IRR
      mirr(fin_rate, reinv_rate): Compute MIRR at given rates
      payback(marr): Compute discounted paybakck period at marr. Project MARR
                       is used if marr is not given.
      is_feasible(marr): Return True or False on project feasible at marr.
                          Project MARR is used if marr is not given.
```

```
[ ]:
```

5.2 Function PnAF_cf

```
[1]: from EngFinancialPy import PnAF_cf
     print(PnAF_cf.__doc__)
```

```
    PnAF_cf(Nper, P=0, A-0, F=0)
      Constructs  [ P, A, A, ..., A , A+F ]
      Parameters:
         Nper = number of periods
         P = Initial cash flow at EoY 0
```

```
      A = Uniform annual cash flow amounts from EoY 1 to EoY n
      F = Single final cash flow at EoY Nper
   Returns a list of cash flows [ P, A, A, ..., A , A+F ]
```

[ ]:

```

```

5.3 Function PnGF_cf

[1]:

```
from EngFinancialPy import PnGF_cf
print(PnGF_cf.__doc__)
```

```
  PnGF_cf(Nper, P=0, A1=0, G=0, F=0):
    Construct [P, A1, A1+G, A1+2G, ..., A1+(N-1)*G + F ] cash flows
    Parameters:
      Nper = Number of periods
      P = Initial case flow
      A1 = Cash flow at EoY 1
      G = Annual cash flows increment up to EoY N
      F = SV at EoY N
  Returns:
    List [P, A1, A1+G, A1+2G, ...,A1+(Nper-2)*G, A1+(Nper-1)*G + F ]
```

[ ]:

```

```

5.4 Examples on Equivalent Worth and Rate of Return Methods

5.4.1 ABC Company

Source: 3.7.2_financial_analysis_ABC_company.ipynb

[1]:

```
# 3.7.2_finanical_analysis_ABC_company.ipynb
""" 3.7.2 Finanical Analysis of ABC Company """
from EngFinancialPy import Project_CF, PnAF_cf
```

[2]:

```
# Create the project cash flows and check basic attributes
ABC = Project_CF(marr=0.2, name="ABC Company Investment Problem")
ABC.set_cf(PnAF_cf(Nper=5, P=-25000, A=8000, F=5000))
print(f"\n{ABC.name}")
print(f"  life = {ABC.life}")
print(f"  Cash flows = {ABC.cf}")
```

```
ABC Company Investment Problem
  life = 5
  Cash flows = [-25000, 8000, 8000, 8000, 8000, 13000]
```

[3]:

```
# Compute Project Profitability Measures
print("Project Profitability Measures:")
print(f"  NPV({ABC.marr}) = {ABC.npv():,.2f}")
print(f"  PW({ABC.marr}) = {ABC.pw():,.2f}")
```

```
print(f"  AW({ABC.marr}) = {ABC.aw():,.2f}")
print(f"  FW({ABC.marr}) = {ABC.fw():,.2f}")
print(f"  IRR = {ABC.irr():.5f}")
```

```
Project Profitability Measures:
  NPV(0.2) = 934.28
  PW(0.2) = 934.28
  AW(0.2) = 312.41
  FW(0.2) = 2,324.80
  IRR = 0.21578
```

[4]:
```
# Compute MIRR at finanical rate = 0.15 and reinvestment rate = 0.20
print(f"  MIRR = {ABC.mirr(fin_rate=0.15, reinv_rate=0.20):8.5f}")
```

```
  MIRR =  0.20884
```

[5]:
```
# Compute liqidity measures
print("Project Liquidity Measure:")
print(f"  Payback({ABC.marr}) = {ABC.payback()}")
```

```
Project Liquidity Measure:
  Payback(0.2) = 5
```

[6]:
```
# Is the project financially feasible?
print("Project Feasibilty:")
print(f"  Feasibility({ABC.marr}) = {ABC.is_feasible()}")
```

```
Project Feasibilty:
  Feasibility(0.2) = True
```

[7]:
```
# Compute the Project PW at marr=10% (instead of default 20%)
print(f"NPV(0.1) = {ABC.npv(0.1):,.2f}")
```

```
NPV(0.1) = 8,430.90
```

[8]:
```
# Compute the Project payback period at marr=10% (instead of 20%)
print(f"Payback(0.1) = {ABC.payback(0.1)}")
```

```
Payback(0.1) = 4
```

[ ]:
```

```

5.4.2 XYZ Company

Source: 3.7.2_financial_analysis_XYZ_company.ipynb

[1]:
```
# 3.7.2_finanical_analysis_XYZ_company.ipynb
""" 3.7.2 Finanical Analysis of XYZ Company """
from EngFinancialPy import Project_CF, PnAF_cf
```

[2]:
```
# Create the project cash flows and check basic attributes
XYZ = Project_CF(marr=0.1, name="XYZ Company Investment Problem")
XYZ.set_cf(PnAF_cf(Nper=5, P=-12000, A=2310, F=1000))
print(f"\n{XYZ.name}")
```

```
print(f"  life = {XYZ.life}")
print(f"  Cash flows = {XYZ.cf}")
```

```
XYZ Company Investment Problem
  life = 5
  Cash flows = [-12000, 2310, 2310, 2310, 2310, 3310]
```

[3]:
```
# Compute project's profitability measures
print("Project Profitability Measures:")
print(f"  NPV({XYZ.marr}) = {XYZ.npv():,.2f}")
print(f"  PW({XYZ.marr}) = {XYZ.pw():,.2f}")
print(f"  AW({XYZ.marr}) = {XYZ.aw():,.2f}")
print(f"  FW({XYZ.marr}) = {XYZ.fw():,.2f}")
print(f"  IRR = {XYZ.irr():.5f}")
```

```
Project Profitability Measures:
  NPV(0.1) = -2,622.36
  PW(0.1) = -2,622.36
  AW(0.1) = -691.77
  FW(0.1) = -4,223.34
  IRR = 0.01436
```

[4]:
```
# Compute MIRR at finanical rate = 0.15 and reinvestment rate = 0.20
print(f"  MIRR = {XYZ.mirr(fin_rate=0.15, reinv_rate=0.20):.5f}")
```

```
  MIRR = 0.08675
```

[5]:
```
# Compute liqidity measures
print("Project Liquidity Measure:")
print(f"  Payback({XYZ.marr}) = {XYZ.payback()}")
```

```
Project Liquidity Measure:
  Payback(0.1) = None
```

[6]:
```
# Is the project finanically feasible?
print("Project Feasibilty:")
print(f"  Feasibility({XYZ.marr}) = {XYZ.is_feasible()}")
```

```
Project Feasibilty:
  Feasibility(0.1) = False
```

[ ]:
```

```

5.4.3 Charlie Company

Source: 3.7.2_financial_analysis_Charlie_company.ipynb

[1]:
```
# 3.7.2_financial_analysis_Charlie_company.ipynb
""" 3.7.2 Financial Analysis of Charlie company """
from EngFinancialPy import Project_CF
```

[2]:
```
# Create the project cash flows and check basic attributes
charlie = Project_CF(marr=0.2, name="Charlie Company Problem")
```

```
charlie.set_cf([-10000, -5000, 5000, 5000, 5000, 5000, 5000])
print(f"\n{charlie.name}")
print(f"  life = {charlie.life}")
print(f"  Cash flows = {charlie.cf}")
```

```
Charlie Company Problem
  life = 6
  Cash flows = [-10000, -5000, 5000, 5000, 5000, 5000, 5000]
```

[3]:
```
# Compute Project Profitability Measures
print("Project Profitability Measures:")
print(f"  NPV({charlie.marr}) = {charlie.npv():,.2f}")
print(f"  PW({charlie.marr}) = {charlie.pw():,.2f}")
print(f"  AW({charlie.marr}) = {charlie.aw():,.2f}")
print(f"  FW({charlie.marr}) = {charlie.fw():,.2f}")
print(f"  IRR = {charlie.irr():.5f}")
```

```
Project Profitability Measures:
  NPV(0.2) = -1,705.78
  PW(0.2) = -1,705.78
  AW(0.2) = -512.94
  FW(0.2) = -5,093.44
  IRR = 0.15529
```

[4]:
```
# Compute MIRR at finanical rate=0.15, reinvestment rate=0.20
print(f"  MIRR = {charlie.mirr(fin_rate=0.15, reinv_rate=0.20):.5f}")
```

```
  MIRR = 0.17213
```

[5]:
```
# Compute Project Liqidity Measures
print("Project Liquidity Measure:")
print(f"Payback({charlie.marr}) = {charlie.payback()}")
```

```
Project Liquidity Measure:
Payback(0.2) = None
```

[6]:
```
# Is the project finanically feasible?
print("Project Feasibilty:")
print(f"  Feasibility({charlie.marr}) = {charlie.is_feasible()}")
```

```
Project Feasibilty:
  Feasibility(0.2) = False
```

[ ]:

5.5 Function CR (Capital Recovery Amount)

[1]:
```
from EngFinancialPy import CR
print(CR.__doc__)
```

```
 Function to Compute and Return Captial Recovery Amount
    Parameters:
      I = Initial investment amount
```

```
SV = Salvage value
rate = marr
N = project life
```

```
[ ]:
```

5.6 Examples on Capital Recovery Amount

Source: 3.2.2_capital_recovery_amount.ipynb

```
[1]: # 3.2.2_capital_recovery_amount.ipynb
     """ 3.2.2 Capital Recovery Amount """
     from EngFinancialPy import CR
```

```
[2]: """ Example (3.2.2)
         Consider an investment on a machine with initial Cost = $10,000,
         salvage vallue = $2,000 and life = 5 years.  What equivaelnt uniform
         annual benfits must the investment provides for it to be financially
         feasible if the marr is 10%
     """
     I = 10_000
     SV = 2_000
     N = 5
     marr = 0.1

     # Using CR function
     cr = CR(I, SV, marr, N)
     print(f"Capital Recovery Amount = {cr:,.2f}")
```

```
Capital Recovery Amount = 2,310.38
```

```
[3]: # Using npf.pmt function directly
     import numpy_financial as npf
     cr = -npf.pmt(marr, N, I, -SV)
     print(f"Capital Recovery Amount = {cr:,.2f}")
```

```
Capital Recovery Amount = 2,310.38
```

```
[ ]:
```

5.7 SubClass pub_Project

```
[1]: from EngFinancialPy import pub_Project
```

```
[2]: print(pub_Project.__doc__)
```

```
Subclass of Project_CF for B/C ratio methods
       Costs and Benefits cash flows are separate inputs
   Attributes:
       benefits_cf : List of benefits cash flows
       costs_cf : List of costs cash flows
   Methods:
```

```
        set_BC_cash_flows(Benfits_CF, Costs_CF)
        BC_Ratio(rate): Computes the BC ratio
```

[ ]:

5.8 Examples on B/C Ratio Methods

5.8.1 Airport Expansion Problem with B/C ratio methods

Source: 3.7.3_financial_analysis_airport_expansion_BC_ratio_methods.ipynb

[1]:
```
# 3.7.3_financial_analysis_airport_expansion_BC_ratio_methods.ipynb
""" 3.7.3 Financial Analysis of Airport Expansion Problem using B/C ratios """
from EngFinancialPy import pub_Project, PnAF_cf
```

[2]:
```
# Project data
capital_costs = -1_200_000
a_benefits    =    490_000
a_disbenefits =   -100_000
a_om_costs    =   -197_500
study_period  =  20
```

[3]:
```
# Create a public project cash flows with benefits and costs cash flows
Airport = pub_Project(marr=0.1, name="Airport expansion problem")
```

[4]:
```
# Set up the Benefits and Costs cash flows for Conventional BC Ratio
B_CF = PnAF_cf(Nper=study_period, A=a_benefits+a_disbenefits)
C_CF = PnAF_cf(Nper=study_period, P=capital_costs, A=a_om_costs)
Airport.set_BC_cash_flows(Benefits_CF=B_CF, Costs_CF=C_CF)
```

[5]:
```
# Compute Conventional B/C ratio
print(f"\n{Airport.name}")
print(f"  life = {Airport.life}")
print(f"  Conventional B/C Ratio = {Airport.BC_Ratio():.4f}")
```

```
Airport expansion problem
  life = 20
  PW of benefits = 3,320,289.85
  PW of costs    = 2,881,428.83
  Conventional B/C Ratio = 1.1523
```

[6]:
```
# Set up the Benefits and Costs cash flows for Modified BC Ratio
B_CF = PnAF_cf(Nper=study_period, A=a_benefits+a_disbenefits+a_om_costs)
C_CF = PnAF_cf(Nper=study_period, P=capital_costs)
Airport.set_BC_cash_flows(Benefits_CF=B_CF, Costs_CF=C_CF)
```

[7]:
```
# Compute Modified B/C ratio
print(f"\n{Airport.name}")
print(f"  life = {Airport.life}")
print(f"  Modified B/C Ratio = {Airport.BC_Ratio():.4f}")
```

```
Airport expansion problem
  life = 20
  PW of benefits = 1,638,861.02
  PW of costs    = 1,200,000.00
  Modified B/C Ratio = 1.3657
```

[8]:
```
# Compute Project DCF Profitability Measures
print("Project Profitability Measures:")
print(f"  PW({Airport.marr}) = {Airport.pw():,.2f}")
print(f"  AW({Airport.marr}) = {Airport.aw():,.2f}")
print(f"  FW({Airport.marr}) = {Airport.fw():,.2f}")
print(f"  IRR = {Airport.irr():.5f}")
```

```
Project Profitability Measures:
  PW(0.1) = 438,861.02
  AW(0.1) = 51,548.45
  FW(0.1) = 2,952,437.46
  IRR = 0.15074
```

[9]:
```
# Compute MIRR at financial rate = 0.1 and reinvestment rate = 0.1
print(f"  MIRR = {Airport.mirr(fin_rate=0.1, reinv_rate=0.1):.5f}")
```

```
  MIRR = 0.11728
```

[10]:
```
# Compute Project Liqidity Measures
print("Project Liquidity Measure:")
print(f"  Payback({Airport.marr}) = {Airport.payback()}")
```

```
Project Liquidity Measure:
  Payback(0.1) = 11
```

[11]:
```
# Is the project financial feasible?
print("Project Feasibilty:")
print(f"  Feasibility({Airport.marr}) = {Airport.is_feasible()}")
```

```
Project Feasibilty:
  Feasibility(0.1) = True
```

6 Financial Decision on Multiple Projects

6.1 Function Evaluate_Projects

[1]:
```
from EngFinancialPy import Evaluate_Projects
```

[2]:
```
print(Evaluate_Projects.__doc__)
```

```
 Evaluate a list of projects using specificed method
    Parameters:
        plist = list of Project_CF objects
        marr = marr to be used for this evaluation
        method = "PW" (default), "AW", "PW", "IRR", "BC_Ratio"
    Return:
```

```
best project
```

```
[ ]:
```

6.2 Projects with Equal Lives: Equivalent Worth Methods

6.2.1 Investment Projects

Source: 4.3.1_investment_projects_equal_lives_equalivant_worth_methods.ipynb

```
[1]: # 4.3.1_investment_projects_equal_lives_equivalent_worth_methods.ipynb
     """ 4.3.1 Investments with Equal Lives - Equivalent Worth methods """
     from EngFinancialPy import Project_CF, PnAF_cf, Evaluate_Projects
```

```
[2]: # Project basic parameters
     marr = 0.1
     study_period = 10
```

```
[3]: # Create the alternatives
     Proj_A = Project_CF(marr=marr, name="Investment A")
     Proj_A.set_cf(PnAF_cf(Nper=study_period, P=-390000, A=69000, F=0))

     Proj_B = Project_CF(marr=marr, name="Investment B")
     Proj_B.set_cf(PnAF_cf(Nper=study_period, P=-920000, A=167000, F=0))

     Proj_C = Project_CF(marr=marr, name="Investment C")
     Proj_C.set_cf(PnAF_cf(Nper=study_period, P=-660000, A=133500, F=0))

     # To be included for investment altnatives evaluation
     Do_nothing = Project_CF(marr=marr, name="Do nothing")
     Do_nothing.set_cf(PnAF_cf(Nper=study_period))
```

```
[4]: # List of alternatives to be evaluated
     Alternatives = [Proj_A, Proj_B, Proj_C, Do_nothing]
```

```
[5]: # Evaluate the alternatives using different equivalent worth methods
     for method in ["PW", "AW", "FW"]:
         best = Evaluate_Projects(Alternatives, marr=marr, method=method)
         print(f"\nChoose alternative {best.name}")
```

```
Using PW method:
  Investment C: PW(0.1) = 160,299.71
  Investment B: PW(0.1) = 106,142.71
  Investment A: PW(0.1) =  33,975.13
  Do nothing: PW(0.1) =       0.00

Choose alternative Investment C

Using AW method:
  Investment C: AW(0.1) =  26,088.04
  Investment B: AW(0.1) =  17,274.24
```

```
  Investment A: AW(0.1) =   5,529.30
  Do nothing: AW(0.1) =       0.00

Choose alternative Investment C

Using FW method:
  Investment C: FW(0.1) = 415,776.16
  Investment B: FW(0.1) = 275,306.85
  Investment A: FW(0.1) =  88,122.74
  Do nothing: FW(0.1) =       0.00

Choose alternative Investment C
```

[ ]:

6.2.2 Cost Projects

Source: 4.3.1_cost_projects_equal_lives_equivalent_worth_methods.ipynb

[1]:
```
# 4.3.1_cost_projects_equal_lives_equivalent_worth_methods.ipynb
""" 4.3.1 Cost Projects with Equal Lives - Equivalent Worth methods """
from EngFinancialPy import Project_CF, PnAF_cf, Evaluate_Projects
```

[2]:
```
# Project basic parameters
marr = 0.1
life = 5
```

[3]:
```
# Create the alternatives
Proj_A = Project_CF(marr=marr, name="Cost Project A")
Proj_A.set_cf(PnAF_cf(Nper=life, P=-24000, A=-31200, F=0))

Proj_B = Project_CF(marr=marr, name="Cost Project B")
Proj_B.set_cf(PnAF_cf(Nper=life, P=-30400, A=-29128, F=0))

Proj_C = Project_CF(marr=marr, name="Cost Project C")
Proj_C.set_cf(PnAF_cf(Nper=life, P=-49600, A=-25192, F=0))

Proj_D = Project_CF(marr=marr, name="Cost Project D")
Proj_D.set_cf(PnAF_cf(Nper=life, P=-52000, A=-22880, F=0))
```

[4]:
```
# List of alternatives to be evaluated
Alternatives = [Proj_A, Proj_B, Proj_C, Proj_D]
```

[5]:
```
# Evaluate the alternatives using different equivalent worth methods
for method in ["PW", "AW", "FW"]:
    best = Evaluate_Projects(Alternatives, marr=marr, method=method)
    print(f"\nChoose alternative {best.name}")
```

```
Using PW method:
  Cost Project D: PW(0.1) = -138,733.20
  Cost Project B: PW(0.1) = -140,818.04
```

```
  Cost Project A: PW(0.1) = -142,272.55
  Cost Project C: PW(0.1) = -145,097.50

Choose alternative Cost Project D

Using AW method:
  Cost Project D: AW(0.1) = -36,597.47
  Cost Project B: AW(0.1) = -37,147.44
  Cost Project A: AW(0.1) = -37,531.14
  Cost Project C: AW(0.1) = -38,276.36

Choose alternative Cost Project D

Using FW method:
  Cost Project D: FW(0.1) = -223,431.21
  Cost Project B: FW(0.1) = -226,788.86
  Cost Project A: FW(0.1) = -229,131.36
  Cost Project C: FW(0.1) = -233,680.98

Choose alternative Cost Project D
```

[ ]:

6.3 Projects with Equal Lives: IRR Method

6.3.1 Investment Projects

Source: 4.3.2_investment_projects_equal_lives_IRR_method.ipynb

[1]:
```
# 4.3.2_investment_projects_equal_lives_IRR_method.ipynb
""" 4.3.2 Investments projects with Equal Lives - IRR Method """
from EngFinancialPy import Project_CF, PnAF_cf, Evaluate_Projects
```

[2]:
```
# Project basic parameters
marr = 0.1
study_period = 10
```

[3]:
```
# Create the Alternatives
Proj_A = Project_CF(marr=marr, name="Investment A")
Proj_A.set_cf(PnAF_cf(Nper=study_period, P=-900, A=150, F=0))

Proj_B = Project_CF(marr=marr, name="Investment B")
Proj_B.set_cf(PnAF_cf(Nper=study_period, P=-1500, A=276, F=0))

Proj_C = Project_CF(marr=marr, name="Investment C")
Proj_C.set_cf(PnAF_cf(Nper=study_period, P=-2500, A=400, F=0))

Proj_D = Project_CF(marr=marr, name="Investment D")
Proj_D.set_cf(PnAF_cf(Nper=study_period, P=-4000, A=925, F=0))

Proj_E = Project_CF(marr=marr, name="Investment E")
Proj_E.set_cf(PnAF_cf(Nper=study_period, P=-5000, A=1125, F=0))
```

```
Proj_F = Project_CF(marr=marr, name="Investment F")
Proj_F.set_cf(PnAF_cf(Nper=study_period, P=-7000, A=1425, F=0))

# To be included for investment alternatives
Do_nothing = Project_CF(marr=marr, name="Do nothing")
Do_nothing.set_cf(PnAF_cf(Nper=study_period))
```

[4]:
```
# List alternatives to be evaluated
Alternatives=[Proj_A, Proj_B, Proj_C, Proj_D, Proj_E, Proj_F, Do_nothing]
```

[5]:
```
# Evaluate the alternatives using IRR method
best = Evaluate_Projects(Alternatives, marr=marr, method="IRR")
print(f"\nBest Alternative is {best.name}")
```

```
Using IRR method:
Sort alternatives by increasing initial costs:
  Do nothing: IRR=nan %
  Investment A: IRR=10.56 %
  Investment B: IRR=12.96 %
  Investment C: IRR=9.61 %
  Investment D: IRR=19.10 %
  Investment E: IRR=18.31 %
  Investment F: IRR=15.57 %

Performing Incremental IRR Analysis:

  base = Do nothing
  next = Investment A
  IRR of increment = 10.56%
  Increment is feasible

  base = Investment A
  next = Investment B
  IRR of increment = 16.40%
  Increment is feasible

  base = Investment B
  next = Investment C
  IRR of increment = 4.12%
  Increment is not feasbile

  base = Investment B
  next = Investment D
  IRR of increment = 22.57%
  Increment is feasible

  base = Investment D
  next = Investment E
  IRR of increment = 15.10%
```

```
  Increment is feasible

  base = Investment E
  next = Investment F
  IRR of increment = 8.14%
  Increment is not feasbile

Best Alternative is Investment E
```

[6]:
```
# Compare the results with PW method
best = Evaluate_Projects(Alternatives, marr=marr, method="PW")
print(f"\nChoose alternative {best.name}")
```

```
Using PW method:
  Investment E: PW(0.1) =   1,912.64
  Investment F: PW(0.1) =   1,756.01
  Investment D: PW(0.1) =   1,683.72
  Investment B: PW(0.1) =     195.90
  Investment A: PW(0.1) =      21.69
  Do nothing: PW(0.1) =       0.00
  Investment C: PW(0.1) =     -42.17

Choose alternative Investment E
```

[ ]:
```

```

6.3.2 Cost Projects

Source: 4.3.2_cost_projects_equal_lives_IRR_method.ipynb

[1]:
```
# 4.3.2_cost_projects_equal_lives_IRR_method.ipynb
""" 4.3.2 Investments projects with Equal Lives - IRR Methods """
from EngFinancialPy import Project_CF, PnAF_cf, Evaluate_Projects
```

[2]:
```
# Project basic parameters
marr = 0.2
study_period = 5
```

[3]:
```
# Create the alternatives
Proj_D1 = Project_CF(marr=marr, name="Cost Project D1")
Proj_D1.set_cf(PnAF_cf(Nper=study_period, P=-100000, A=-29000, F=10000))

Proj_D2 = Project_CF(marr=marr, name="Cost Project D2")
Proj_D2.set_cf(PnAF_cf(Nper=study_period, P=-140600, A=-16900, F=14000))

Proj_D3 = Project_CF(marr=marr, name="Cost Project D3")
Proj_D3.set_cf(PnAF_cf(Nper=study_period, P=-148200, A=-14800, F=25600))

Proj_D4 = Project_CF(marr=marr, name="Cost Project D4")
Proj_D4.set_cf(PnAF_cf(Nper=study_period, P=-122000, A=-22100, F=14000))
```

[4]:
```
# List of alternatives to be evaluated
Alternatives = [Proj_D1, Proj_D2, Proj_D3, Proj_D4]
```

[5]:
```
# Evaluate the alternatives using IRR method
best = Evaluate_Projects(Alternatives, marr=marr, method="IRR")
print(f"\nChoose alternative {best.name}")
```

```
Using IRR method:
Sort alternatives by increasing initial costs:
  Cost Project D1: IRR=nan %
  Cost Project D4: IRR=nan %
  Cost Project D2: IRR=nan %
  Cost Project D3: IRR=-60.60 %

Performing Incremental IRR Analysis:

  base = Cost Project D1
  next = Cost Project D4
  IRR of increment = 20.47%
  Increment is feasible

  base = Cost Project D4
  next = Cost Project D2
  IRR of increment = 12.31%
  Increment is not feasbile

  base = Cost Project D4
  next = Cost Project D3
  IRR of increment = 20.44%
  Increment is feasible

Choose alternative Cost Project D3
```

[6]:
```
# Compare the results with PW method
best = Evaluate_Projects(Alternatives, marr=marr, method="PW")
print(f"\nChoose alternative {best.name}")
```

```
Using PW method:
  Cost Project D3: PW(0.2) = -182,172.99
  Cost Project D4: PW(0.2) = -182,466.24
  Cost Project D1: PW(0.2) = -182,708.98
  Cost Project D2: PW(0.2) = -185,515.06

Choose alternative Cost Project D3
```

[ ]:
```
```

6.4 Projects with Equal Lives B/C Ratio Methods

6.4.1 Investment Projects

Source 4.3.3_investment_projects_equal_lives_BC_ratio_methods.ipynb

[1]:
```
# 4.3.3_investment_projects_equal_lives_BC_ratio_methods.ipynb
""" Investment Projects with Equal Lives B/C ratio methods """
from EngFinancialPy import pub_Project, PnAF_cf, Evaluate_Projects
```

[2]:
```
# Project basic parameters
marr = 0.1
study_period = 50
```

[3]:
```
# Create the alternatives
Proj_A = pub_Project(marr=marr, name="Project A")
Proj_A.set_BC_cash_flows(
    Benefits_CF=PnAF_cf(Nper=study_period, A=2150000),
    Costs_CF=PnAF_cf(Nper=study_period,P=-8500000, A=-750000, F=1250000))

Proj_B = pub_Project(marr=marr, name="Project B")
Proj_B.set_BC_cash_flows(
    Benefits_CF=PnAF_cf(Nper=study_period, A=2265000),
    Costs_CF=PnAF_cf(Nper=study_period,P=-10000000, A=-725000, F=1750000))

Proj_C = pub_Project(marr=marr, name="Project C")
Proj_C.set_BC_cash_flows(
    Benefits_CF=PnAF_cf(Nper=study_period, A=2500000),
    Costs_CF=PnAF_cf(Nper=study_period,P=-12000000, A=-700000, F=2000000))

Do_nothing = pub_Project(marr=marr, name="Do nothing")
Do_nothing.set_BC_cash_flows(
    Benefits_CF=PnAF_cf(Nper=study_period),
    Costs_CF=PnAF_cf(Nper=study_period))
```

[4]:
```
# List of alternatives to be evaluated
Alternatives = [Proj_A, Proj_B, Proj_C, Do_nothing]
```

[5]:
```
# Evaluate the alternatives using BC Ratio method
best = Evaluate_Projects(Alternatives, marr=marr, method="BC_Ratio")
print(f"\nChoose alternative {best.name}")
```

```
Using BC Ratio method:
Sort alternatives by increasing PW of costs:
  Do nothing:  PW of Costs = -0.00
  Project A :  PW of Costs = 15,925,462.68
  Project B :  PW of Costs = 17,173,333.04
  Project C :  PW of Costs = 18,923,333.04

Performing Incremental B/C Ratio Analysis:

  base = Do nothing
```

```
  next = Project A
  BC ratio of increment = 1.3385
  Increment is feasible

  base = Project A
  next = Project B
  BC ratio of increment = 0.9137
  Increment is not feasible

  base = Project A
  next = Project C
  BC ratio of increment = 1.1576
  Increment is feasible

Choose alternative Project C
```

[6]:
```
# Compare the results with PW method
best = Evaluate_Projects(Alternatives, marr=marr, method="PW")
print(f"\nChoose alternative {best.name}")
```

```
Using PW method:
  Project C : PW(0.1) = 5,863,703.18
  Project A : PW(0.1) = 5,391,388.47
  Project B : PW(0.1) = 5,283,721.78
  Do nothing: PW(0.1) =        0.00

Choose alternative Project C
```

[ ]:
```

```

6.5 Projects with Unequal Lives Repeatability Assumption

6.5.1 Investment Projects

Source: 4.4.1_investment_projects_unequal_lives_repeatability.ipynb

[1]:
```
# 4.4.1_investment_projects_unequal_lives_repeatability.ipynb
""" 4.4.1 Investment projects with unequal live - Repeatability """
from EngFinancialPy import Project_CF, PnAF_cf, Evaluate_Projects
```

[2]:
```
# Basic project parameters
marr = 0.1
study_period = 12
```

[3]:
```
# Create the alternatives
Proj_A = Project_CF(marr=marr, name="Investment A")
Proj_A.set_cf(PnAF_cf(Nper=4, P=-3500, A=1900-645, F=0))

Proj_B = Project_CF(marr=marr, name="Investment B")
Proj_B.set_cf(PnAF_cf(Nper=6, P=-5000, A=2500-1020, F=0))
```

[4]:
```
# List of alternatives to be evaluated
Alternatives = [Proj_A, Proj_B]
```

[5]:
```
# Evaluate the alternatives using AW method under repeability assumption
best = Evaluate_Projects(Alternatives, marr=marr, method="AW")
print(f"\nChoose alternative {best.name} under repeatability assumption",
      f"with study period {study_period} years")
```

```
Using AW method:
  Investment B: AW(0.1) =     331.96
  Investment A: AW(0.1) =     150.85

Choose alternative Investment B under repeatability assumption with study period
12 years
```

[ ]:
```

```

6.5.2 Cost Projects

Source: 4.4.2_cost_projects_unequal_lives_repeatability.ipynb

[1]:
```
# 4.4.2_cost_projects_unequal_lives_repeatability.ipynb
""" 4.4.2 Pump selection problem under repeatability assumption """
from EngFinancialPy import Project_CF, Evaluate_Projects
```

[2]:
```
# Pump selection problem - Repeatability assumption
# Basic project parameters
marr = 0.2
study_period = 45
```

[3]:
```
# Create the Alternatives
SP240 = Project_CF(marr=marr, name="SP240")
cap_cost = -33200
e_cost = -2165
m_costY1= -1100
m_inc = -500
sv5 = 0
SP240.set_cf([cap_cost,
              e_cost + m_costY1,
              e_cost + m_costY1 + m_inc,
              e_cost + m_costY1 + m_inc*2,
              e_cost + m_costY1 + m_inc*3,
              e_cost + m_costY1 + m_inc*4 + sv5 ])

HEPS9 = Project_CF(marr=marr, name="HEPS9")
cap_cost = -47600
e_cost = -1720
m_costY4= -500
m_inc = -100
sv9 = 5000
HEPS9.set_cf([cap_cost,
```

```
                e_cost, e_cost, e_cost,
                e_cost + m_costY4,
                e_cost + m_costY4 + m_inc*1,
                e_cost + m_costY4 + m_inc*2,
                e_cost + m_costY4 + m_inc*3,
                e_cost + m_costY4 + m_inc*4,
                e_cost + m_costY4 + m_inc*5 + sv9 ])
```

[4]:
```
# List of alternatives to be evaluated
Alternatives = [SP240, HEPS9]
```

[5]:
```
# Evaluate the alternatives using AW method under repeatability assumption
best = Evaluate_Projects(Alternatives, marr=marr, method="AW")
print(f"\nChoose pump {best.name} using AW method under repeatability",
      f"assumption with a study period of {study_period} years")
```

```
Using AW method:
  HEPS9      : AW(0.2) = -13,621.37
  SP240      : AW(0.2) = -15,186.66

Choose pump HEPS9 using AW method under repeatability assumption with a study
period of 45 years
```

[ ]:
```

```

6.6 Projects with Unequal Lives Cotermination Assumption

6.6.1 Investment Projects with Reinvestment at MARR

Source: 4.5.1_investments_unequal_lives_cotermination_reinvest_marr.ipynb

[1]:
```
# 4.5.1_investments_unequal_lives_cotermination_reinvest_marr.ipynb
""" 4.5.1 Investment projects with unequal live under cotermination
    and reinvestment at marr assumption """
from EngFinancialPy import Project_CF, PnAF_cf, Evaluate_Projects
```

[2]:
```
# Project basic parameters
marr = 0.1
```

[3]:
```
# Create the alternatives
Proj_A = Project_CF(marr=marr, name="Investment A")
Proj_A.set_cf(PnAF_cf(Nper=4, P=-3500, A=1900-645, F=0))

Proj_B = Project_CF(marr=marr, name="Investment B")
Proj_B.set_cf(PnAF_cf(Nper=6, P=-5000, A=2500-1020, F=0))
```

[4]:
```
# List of alternatives to be evaluated
Alternatives = [Proj_A, Proj_B]
```

[5]:
```
# Evaluate the alternatives using PW method under cotermination at
# year 6 with reinvestment at marr assumption
```

```
best = Evaluate_Projects(Alternatives, marr=marr, method="PW")
print(f"\nChoose alternative {best.name} under co-termination at EoY 6",
    "with reinvestment at marr")
```

```
Using PW method:
  Investment B: PW(0.1) =   1,445.79
  Investment A: PW(0.1) =     478.18

Choose alternative Investment B under co-termination at EoY 6 with reinvestment
at marr
```

[ ]:

6.6.2 Cost Projects: Forklift Truck Replacement Problem

Source: 4.5.2_cost_projects_(forklift)_unequal_lives_cotermination.ipynb

[1]:

```
# 4.5.2_cost_projects_(forklift)_unequal_lives_cotermination.ipynb
""" 4.5.2 Cost projects (forklift) with unequal lives under
    cotermination assumption """
from EngFinancialPy import Project_CF, Evaluate_Projects
```

[2]:

```
# Forklift Truck Selection Problem undeer cotermination at EoY 9
# Project basic parameters
marr = 0.15
study_period = 8
```

[3]:

```
# Create the alternatives
StackHigh = Project_CF(marr=marr, name="Stackhigh")
capital_cost = -184000
annual_cost = -30000
sv5 = 17000
lifeSH = 5
lease3Y = -104000
StackHigh.set_cf([capital_cost] +
                 [annual_cost]*(lifeSH-1) +
                 [annual_cost + sv5] +
                 [lease3Y]*3)

S2000 = Project_CF(marr=marr, name="S2000")
capital_cost = -242000
annual_cost = -26700
sv7 = 21000
lifeS2k = 7
lease1Y = -134000
S2000.set_cf([capital_cost] +
             [annual_cost]*(lifeS2k-1) +
             [annual_cost + sv7] +
             [lease1Y])
```

```
[4]: # List of alternatives to be evaluated
Alternatives = [StackHigh, S2000]
```

```
[5]: # Evaluate the alternatives using PW method under cotermination assumption
best = Evaluate_Projects(Alternatives, marr=marr, method="PW")
print(f"\nChoose folklift {best.name} under cotermination at",
      f"EoY {study_period}")
```

```
Using PW method:
  S2000     : PW(0.15) = -388,993.37
  Stackhigh : PW(0.15) = -394,169.96

Choose folklift S2000 under cotermination at EoY 8
```

```
[6]: # Evaluate the alternatives using IRR method under cotermination assumption
best = Evaluate_Projects(Alternatives, marr=marr, method="IRR")
print(f"\nChoose folklift {best.name} under cotermination at",
      f"EoY {study_period}")
```

```
Using IRR method:
Sort alternatives by increasing initial costs:
  Stackhigh: IRR=nan %
  S2000: IRR=nan %

Performing Incremental IRR Analysis:

  base = Stackhigh
  next = S2000
  IRR of increment = 16.70%
  Increment is feasible

Choose folklift S2000 under cotermination at EoY 8
```

```
[ ]:
```

6.6.3 Cost Projects: Pump Replacement Problem

Source: 4.5.2_cost_projects_(pump_selection)_unequal_lives_cotermination.ipynb

```
[1]: # 4.5.2_cost_projects_(pump_selection)_unequal_lives_cotermination.ipynb
""" 4.5.2 Cost projects (pump selection) with unequal lives under
    cotermination assumption """
from EngFinancialPy import Project_CF, Evaluate_Projects
```

```
[2]: # Pump Selection Problem undeer cotermination at EoY 5
# Project basic parameters
marr = 0.2
study_period = 5
```

[3]:
```python
# Create the alternatives
SP240 = Project_CF(marr=marr, name="SP240")
cap_cost = -33200
e_cost = -2165
m_costY1= -1100
m_inc = -500
sv5 = 0
SP240.set_cf([cap_cost,
              e_cost + m_costY1,
              e_cost + m_costY1 + m_inc,
              e_cost + m_costY1 + m_inc*2,
              e_cost + m_costY1 + m_inc*3,
              e_cost + m_costY1 + m_inc*4 + sv5 ])

HEPS9 = Project_CF(marr=marr, name="HEPS9")
cap_cost = -47600
e_cost = -1720
m_costY4= -500
m_inc = -100
sv5 = 15000
HEPS9.set_cf([cap_cost,
              e_cost, e_cost, e_cost,
              e_cost + m_costY4,
              e_cost + m_costY4 + m_inc*1 + sv5])
```

[4]:
```python
# List of alternatives to be evaluated
Alternatives = [SP240, HEPS9]
```

[5]:
```python
# Evaluate the alternatives using PW method under cotermination assumption
best = Evaluate_Projects(Alternatives, marr=marr, method="PW")
print(f"\nChoose pump {best.name} under cotermination assumption",
      f"at EoY {study_period}")
```

```
Using PW method:
  SP240      : PW(0.2) = -45,417.41
  HEPS9      : PW(0.2) = -47,197.94

Choose pump SP240 under cotermination assumption at EoY 5
```

[6]:
```python
# Evaluate the alternatives using AW method under cotermination assumption
best = Evaluate_Projects(Alternatives, marr=marr, method="AW")
print(f"\nChoose pump {best.name} under cotermination assumption",
      f"at EoY {study_period}")
```

```
Using AW method:
  SP240      : AW(0.2) = -15,186.66
  HEPS9      : AW(0.2) = -15,782.03

Choose pump SP240 under cotermination assumption at EoY 5
```

```
[ ]:
```

7 Understanding Key Uncertainty using Sensitivity Analysis

7.1 One-Way Range Sensitivity Analysis

7.1.1 Class OneWayRangeSensit

```
[1]: from EngFinancialPy import OneWayRangeSensit
     print(OneWayRangeSensit.__doc__)
```

```
Class for performing one-way range sensitivity analysis
   OneWayRangeSensit(alternatives, range_data, name="Unnamed",
                output_label="$NPV"):
   Parameters:
     alternatives : dictionary of alternatives and objective functions
     range_data : dictionary of variable names and low, base, high values
     name : optional string, default = "Unnamed"
     output_label : optional string, default = "$NPV"

   Attributes:
     name = name of this analysis
     Alternatives = Dictionary of alternatives and objective functions
     Alt_names = list of alternative names
     Obj_functions = list of objective functions
     Var_data = Dictionary of variable names and low, base, high values
     Var_names = list of variable names
     output_label = label for the objective function outputs

  Methods:
     sensit(show_tables=True, show_tornados=True, show_spiders=True,
            precision=4)
       Perform one-way range sensitivity for each alternative,
       dhow sensitivty tables, and plot tornado and spider diagrams

   combined_tornados():
       Plot a combined tornado diagram for all alternatives.
```

```
[ ]:
```

7.1.2 One-Way Range Sensitivty with Tornado & Spider Diagrams

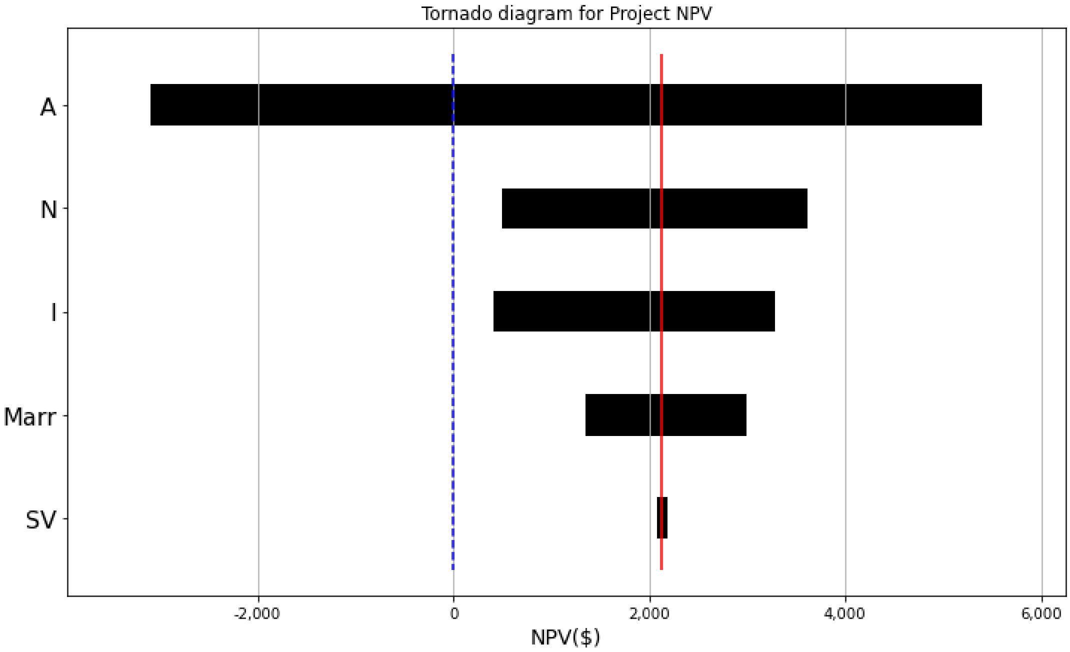
Source: 5.2_one_way_range_sensitivity_analysis.ipynb

```
[1]: # 5.2_one_way_range_sensitivty_analysis.py
     """ One way range sensitivty analysis """
     from EngFinancialPy import OneWayRangeSensit
     import numpy_financial as npf
```

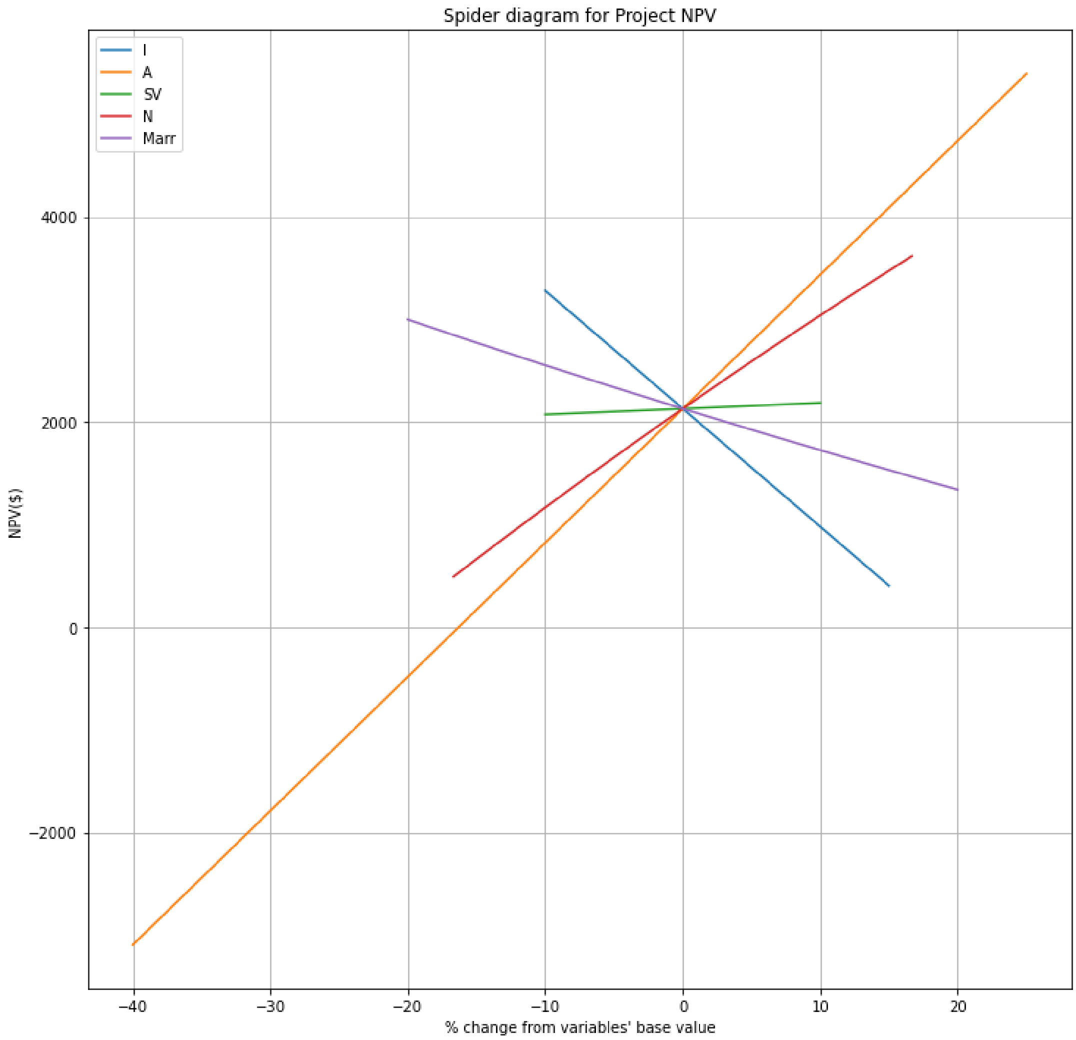
[2]:
```python
# Define the objective functions for the alternatives
# Arguments must all be in the same order
def NPV(I, A, SV, N, rate):
    return I - npf.pv(rate, N, A, SV)
def DoNothing(I, A, SV, N, rate):
    return 0

# Put the alternative names and objective functions in a dictionary
Alternatives = {"Project NPV" : NPV,
                "Do nothing" : DoNothing }

# Put the variable names and the low, base, high values in a dictionary
#           var name:      low,    base,    high values
Var_data = {'I'    : [-13225, -11500, -10350],
            'A'    : [  1800,   3000,   3750],
            'SV'   : [   900,   1000,   1100],
            'N'    : [     5,      6,      7],
            'Marr': [  0.08,     0.1,   0.12]  }

# Label for the objective function outputs
output_label = "NPV($)"

# Name of this analysis
name = "One-Way Range Sensitivty Analysis Example (5.2)"

# Create a problem instance
Project = OneWayRangeSensit(Alternatives, Var_data, name, output_label)
```

[3]:
```python
# Generate individual tables and tornados first
Project.sensit(show_tables=True, show_tornados=True, show_spiders=True,
               precision=2)
```

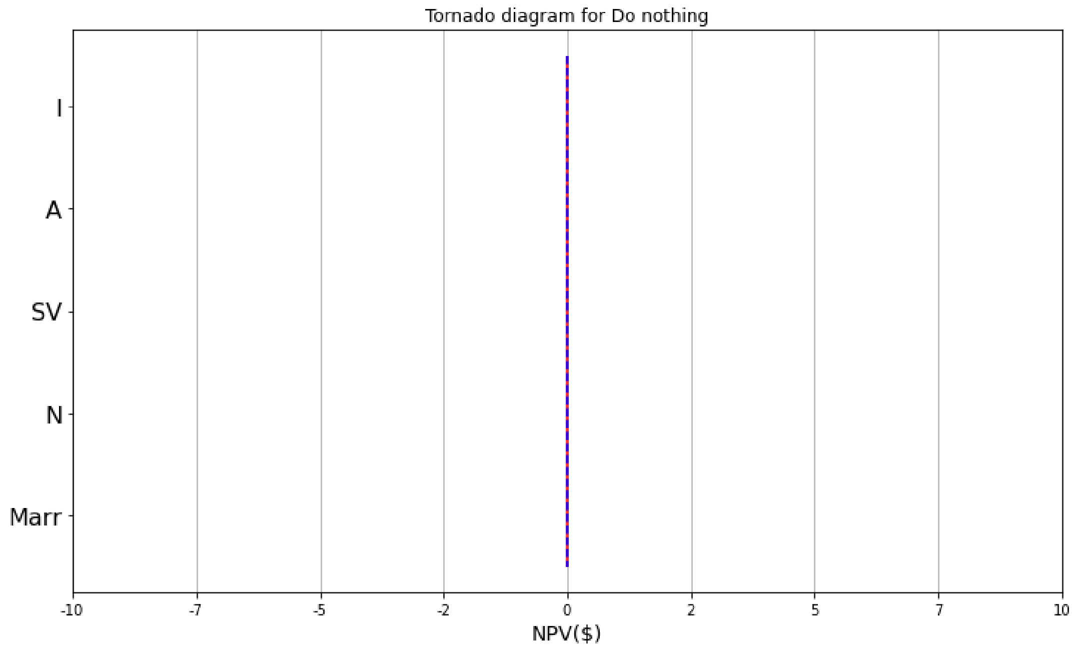
```
One-Way Range Sensitivity for One-Way Range Sensitivty Analysis Example (5.2)

Alternative: Project NPV
            low        base        high   obj_low  obj_high    swing
I    -13,225.00 -11,500.00 -10,350.00    405.26  3,280.26 2,875.00
A      1,800.00   3,000.00   3,750.00 -3,096.06  5,396.70 8,492.76
SV       900.00   1,000.00   1,100.00  2,073.81  2,186.70   112.89
N          5.00       6.00       7.00    493.28  3,618.41 3,125.13
Marr       0.08       0.10       0.12  1,340.85  2,998.81 1,657.96

  Base NPV($) =  2,130.26
  Min  NPV($) =  -3,096.06
  Max  NPV($) =  5,396.70

 Tornado diagram for Project NPV:
```



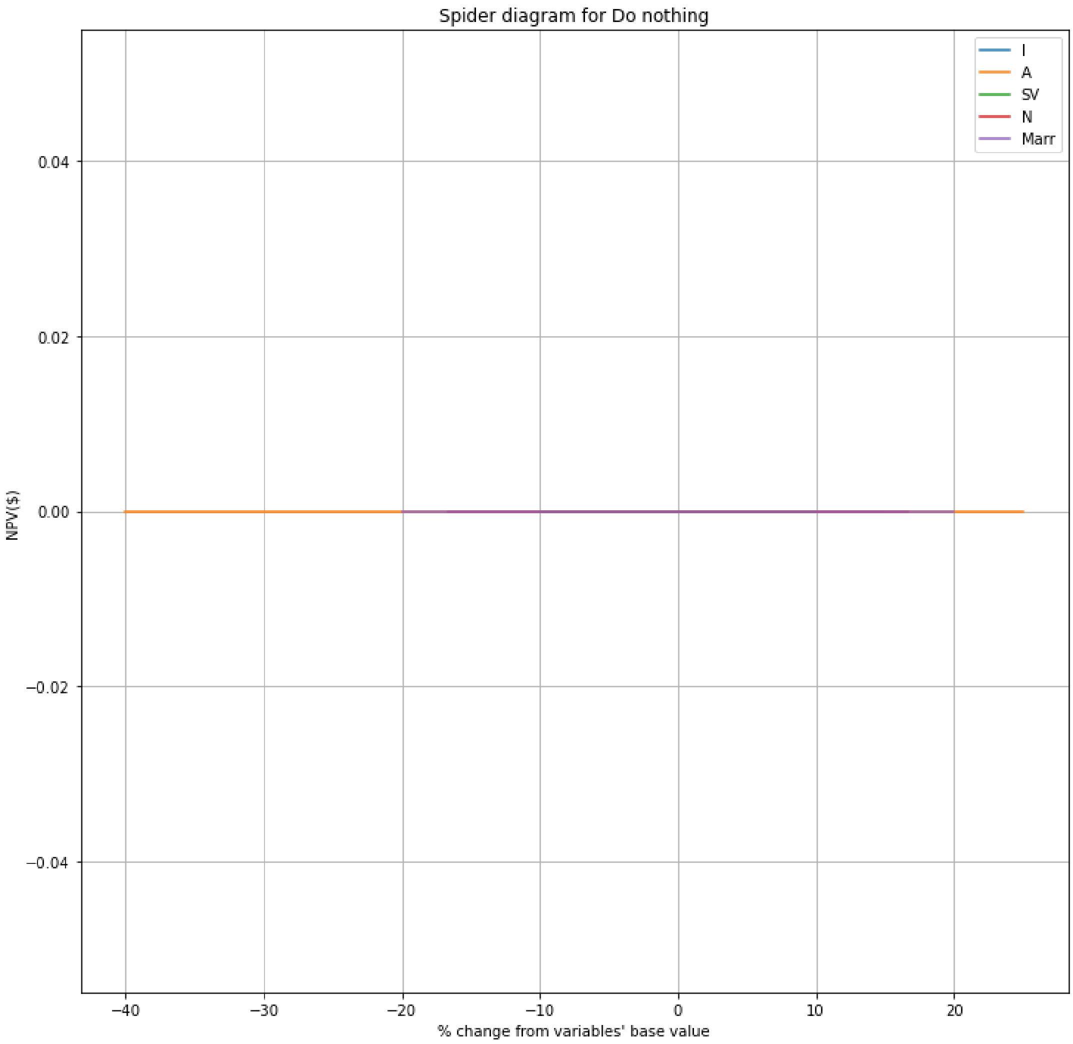
`Spider diagram for Project NPV:`




```
Alternative: Do nothing
             low       base       high  obj_low  obj_high  swing
I     -13,225.00 -11,500.00 -10,350.00        0         0      0
A       1,800.00   3,000.00   3,750.00        0         0      0
SV        900.00   1,000.00   1,100.00        0         0      0
N           5.00       6.00       7.00        0         0      0
Marr        0.08       0.10       0.12        0         0      0

  Base NPV($) =  0.00
  Min  NPV($) =  0.00
  Max  NPV($) =  0.00

 Tornado diagram for Do nothing:
```

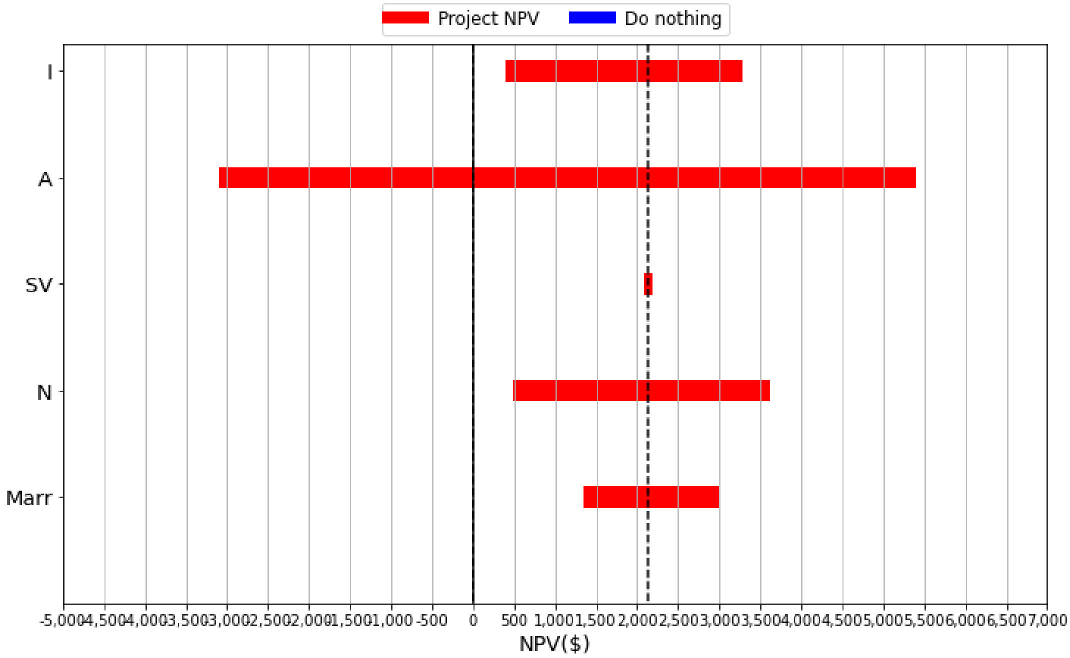


Spider diagram for Do nothing:



[4]:
```
# Plot combined tornados for all alternatives
Project.combined_tornados(-5000, 7000, 500)
```

Combined Tornado Diagrams for One-Way Range Sensitivty Analysis Example (5.2)




```
[ ]:
```

7.2 Break-Even Analysis with Rainbow Diagrams

7.2.1 Class RainbowDiagram

```
[1]: from EngFinancialPy import RainbowDiagram
     print(RainbowDiagram.__doc__)
```

```
Plot Rainbow diagrams and find the break points
   RainbowDiagram(Functions, Names, XL, XH, XStep)
   Parameters:
     Functions :  List of functions to plot
     Names : List of names of functions to plot
     XL : Lower x-axis limit of rainbow diagram
     xH : Upper x-axis limit of rainbow diagram
     xStep : Step size of x-axis
   Methods:
     plot(xL, xH, xStep, xlabel, ylabel, nPoints, dpi)
       Plot Rainbow Diagram
       Parameters:
           XL : Lower x-axis limit of rainbow diagram
           xH : Upper x-axis limit of rainbow diagram
           xStep : Step size of x-axis
           xlabel : x-axis label (default None)
           ylabel : y-axis label (default None)
           nPoints : number of points used to plot diagram (default 100)
           dpi : DPI of point (default 100)

     break_point()
```

```
        Compute the break-even points.
        Parameter:
            Nil
```

[ ]:

```

```

7.2.2 Break-Even and Rainbow Diagram Example

Source: 5.3_break_even_analysis_rainbow_diagrams.ipynb

[1]:

```
#  5.3_break_even_analysis_rainbow_diagrams.ipynb
""" Break-Even Analsyis using Rainbow Diagrams """
from EngFinancialPy import RainbowDiagram
import numpy_financial as npf
```

[2]:

```
# Study period = 10 years
# Repeatability assumption
marr = 0.12
```

[3]:

```
# Alterative I: Auto feed machine
Init_cost_A = -50_000
Life_A = 10
MV10_A = 8_000
OM_A = -7_000
Labor_cost_A = -1*12  # per hour
Output_A = 8  # unit per hour
def auto(q):
    # AW in $K of auto machine for q units per year
    aw = OM_A + q*Labor_cost_A/Output_A - \
        npf.pmt(marr, Life_A, Init_cost_A, MV10_A)
    return aw/1000
```

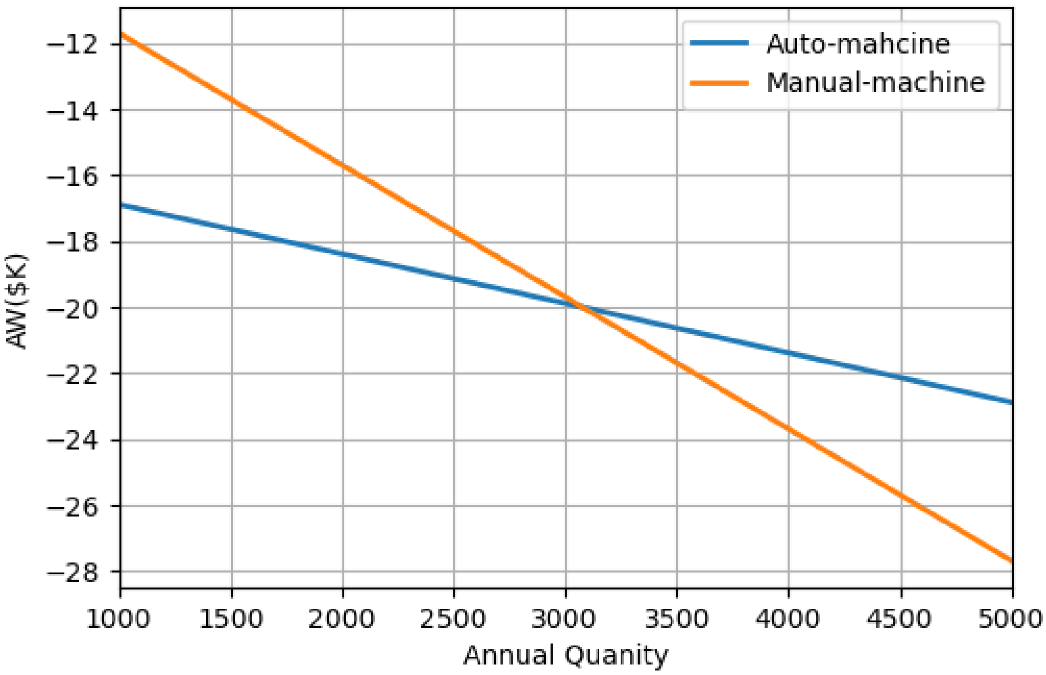
[4]:

```
# Alternative II:  Manual feed machine
Init_cost_M = -17_500
Life_M = 5  # repeatable at EoY 5
MV5_M = 1_000
OM_M = -3_000
Labor_cost_M = -3*8  # per hour
Output_M = 6  # unit per hour
def manual(q):
    # AW in $K of manual machine for q units per year
    aw = OM_M + q*Labor_cost_M/Output_M - \
        npf.pmt(marr, Life_M, Init_cost_M, MV5_M)
    return aw/1000
```

[5]:

```
# The list of functions we want to plot
Fns = [auto, manual]
Names = ['Auto-mahcine', 'Manual-machine']
```

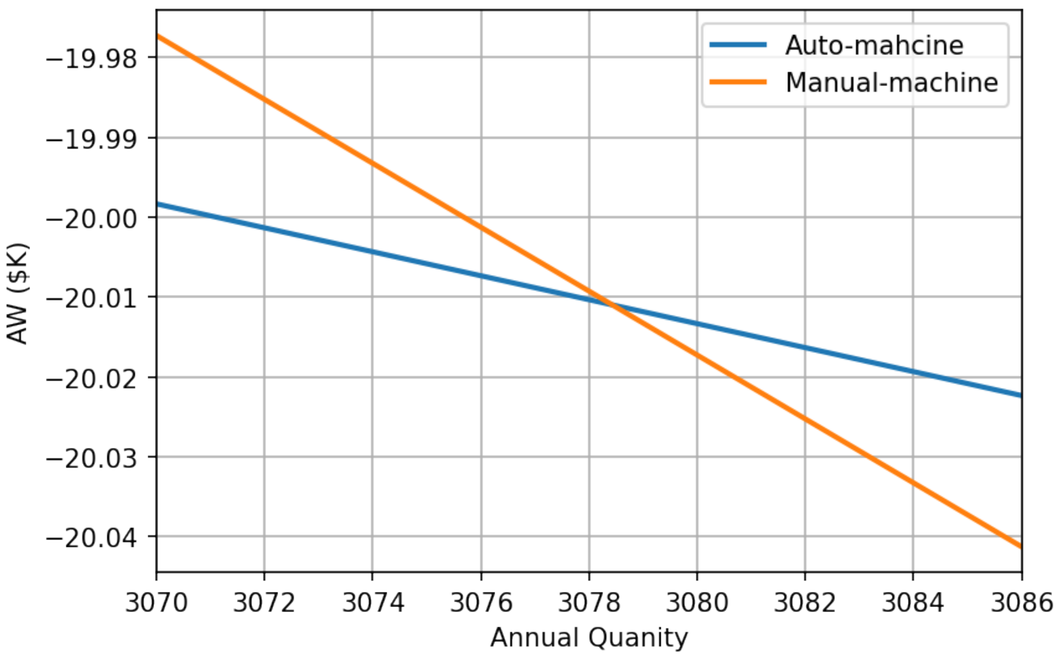
```python
# Create and plot the rainbow diagrams
RB = RainbowDiagram(Fns, Names, xL=1000, xH=5000, xStep=500)
RB.plot(xlabel="Annual Quanity", ylabel='AW($K)')
```



[6]:
```python
# Compute the break-even points
break_even_points = RB.break_points()
print(sorted(break_even_points))
```

```
[3078.429727074458]
```

[7]:
```python
# Zoom-in the rainbow diagram around the break-even point.
RB.plot(xL=3070, xH=3086, xStep=2, xlabel="Annual Quanity",
        ylabel='AW ($K)', dpi=150)
```



[ ]:
```

```

7.3 Decision Reversal & Critical Factors Analysis

Source: 5.4_decision_reversal_critical_factors_analysis.ipynb

[1]:
```
# 5.4_decision_reversal_critical_factors_analysis.ipynb
""" 5.4 Decision Reversal and Critical Factors Analysis """
import numpy as np
import numpy_financial as npf
from EngFinancialPy import IntFactor
import matplotlib.pyplot as plt
from scipy.optimize import root
```

[2]:
```
# Common data
marr = 0.12
Life = 10
```

[3]:
```
# Alternative A
I_A = 170000
R_A = 35000
E_A = 3000
SV_A = 20000

# Alternative B
I_B = 120000
R_B = 40000
EBy1 = 2000
EBinc = 2500
```

```
SV_B = 0
```

[4]:
```
# The objective functions
def PW_A(N, I, R, E, SV):
    """ Compute the PW of Investment A """
    return -I - npf.pv(marr, N, R-E, SV )
def PW_B(N, I, R, SV):
    """ Compute the PW of Investment B"""
    E_B = EBy1 + EBinc*IntFactor('A','G', marr, N).value
    return -I - npf.pv(marr, N, R - E_B, SV )
```

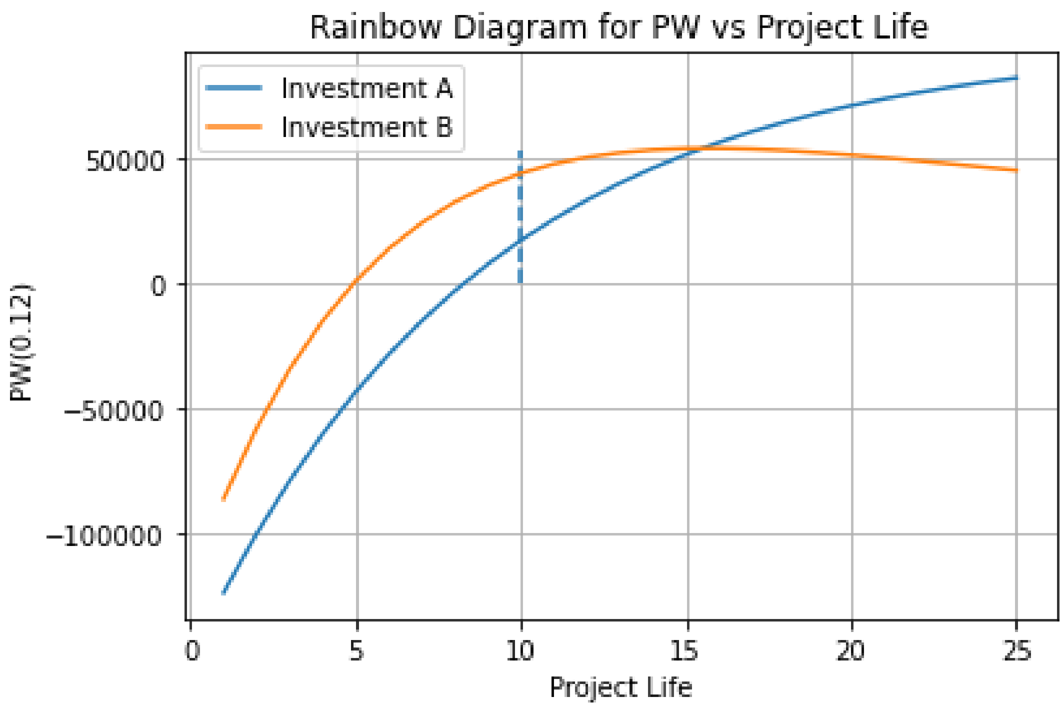
[5]:
```
""" Base Case Analysis """
PW_A_base= PW_A(Life, I_A, R_A, E_A, SV_A)
PW_B_base= PW_B(Life, I_B, R_B, SV_B)
print("Base Case Solutions:")
print(f"  PW_A({marr}) = {PW_A_base:,.2f}")
print(f"  PW_B({marr}) = {PW_B_base:,.2f}")
```

```
Base Case Solutions:
  PW_A(0.12) = 17,246.60
  PW_B(0.12) = 44,073.25
```

[6]:
```
""" Sensitivity Analysis on Project Life """
print("\nSensitivity Analysis on Project Life")

def PW_A_life(n):
    return PW_A(n, I_A, R_A, E_A, SV_A)
def PW_B_life(n):
    return PW_B(n, I_B, R_B, SV_B)

# Plot rainbow diagram
n = np.linspace(1, 25, 25)
f1, ax1 = plt.subplots()
ax1.plot(n, PW_A_life(n), label="Investment A")
ax1.plot(n, PW_B_life(n), label="Investment B")
ax1.vlines(Life, 0, 1.2*max(PW_A_life(Life),PW_B_life(Life)), ls='--')
ax1.legend()
ax1.set_title("Rainbow Diagram for PW vs Project Life")
ax1.set_xlabel("Project Life")
ax1.set_ylabel(f"PW({marr})")
ax1.grid()
plt.show()

# Find break points
guess = 5
# Solve PW_B(n) = 0
bp1 = root(PW_B_life, guess, tol=1e-10).x
print(f"\nBreak point 1 = {bp1[0]:.2f}")
# Solve PW_B(n) = PW_A(n)
bp2 = root(lambda n: PW_A_life(n) - PW_B_life(n), guess, tol=1e-10).x
print(f"Break point 2 = {bp2[0]:.2f}")
```

```
print(f"Change in value required for decision reversal = {bp2[0]-Life:,.2f}")
print(f"%-change = {100*(bp2[0]-Life)/Life:.2f}%" )
```

```
Sensitivity Analysis on Project Life
```

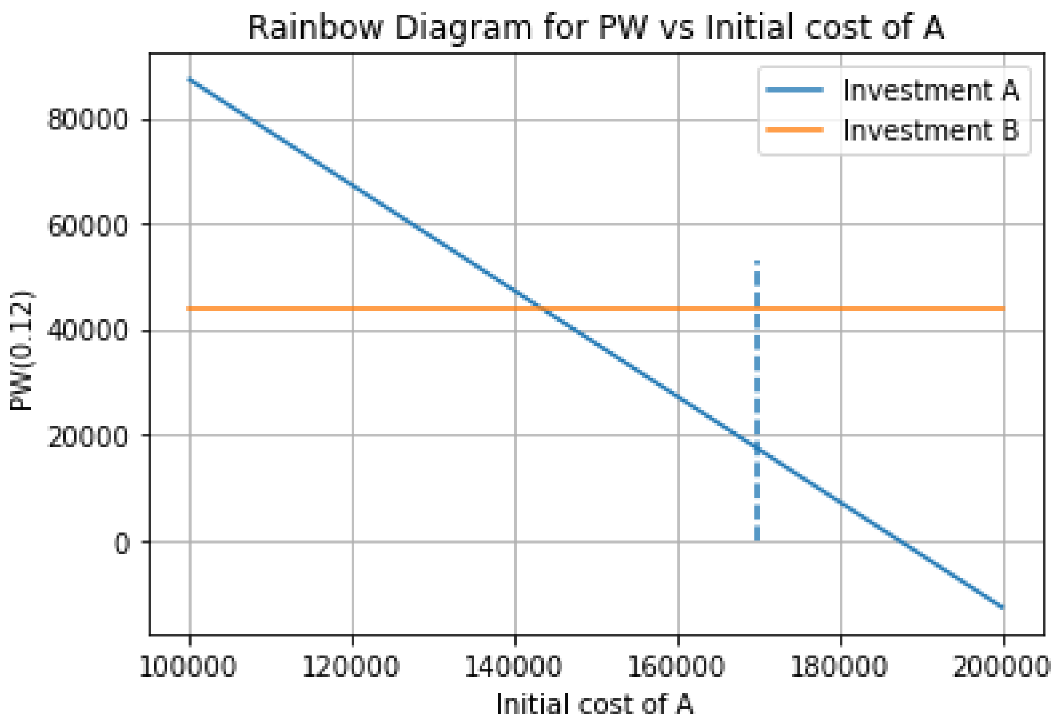



```
Break point 1 = 4.93
Break point 2 = 15.50
Change in value required for decision reversal = 5.50
%-change = 55.04%
```

[7]:
```
""" Sensitivity analysis on Initial cost of Investment A """
print("\nSensitivity Analysis on Initial Cost of Investment A")
def PW_A_I(I):
    return PW_A(Life, I, R_A, E_A, SV_A)

# Plot rainbow diagram
x = np.linspace(100000, 200000, 101)
f2, ax2 = plt.subplots()
ax2.plot(x, PW_A_I(x), label='Investment A')
ax2.plot(x, [PW_B_base]*len(x), label='Investment B')
ax2.vlines(I_A, 0, 1.2*max(PW_A_I(I_A), PW_B_base),ls='--')
ax2.legend()
ax2.set_title("Rainbow Diagram for PW vs Initial cost of A")
ax2.set_xlabel("Initial cost of A")
ax2.set_ylabel(f"PW({marr})")
ax2.grid()
```

```
plt.show()

# Find break point
guess = 140000
# solve PW_A(x)=PW_B_base
bp = root(lambda x: PW_A_I(x)-PW_B_base, guess, tol=1e-10).x
print(f"\nReveral point = {bp[0]:,.2f}")
print(f"Change in value required for decision reversal = {bp[0]-I_A:,.2f}")
print(f"%-change = {100*(bp[0]-I_A)/I_A:.2f}%")
```

```
Sensitivity Analysis on Initial Cost of Investment A
```

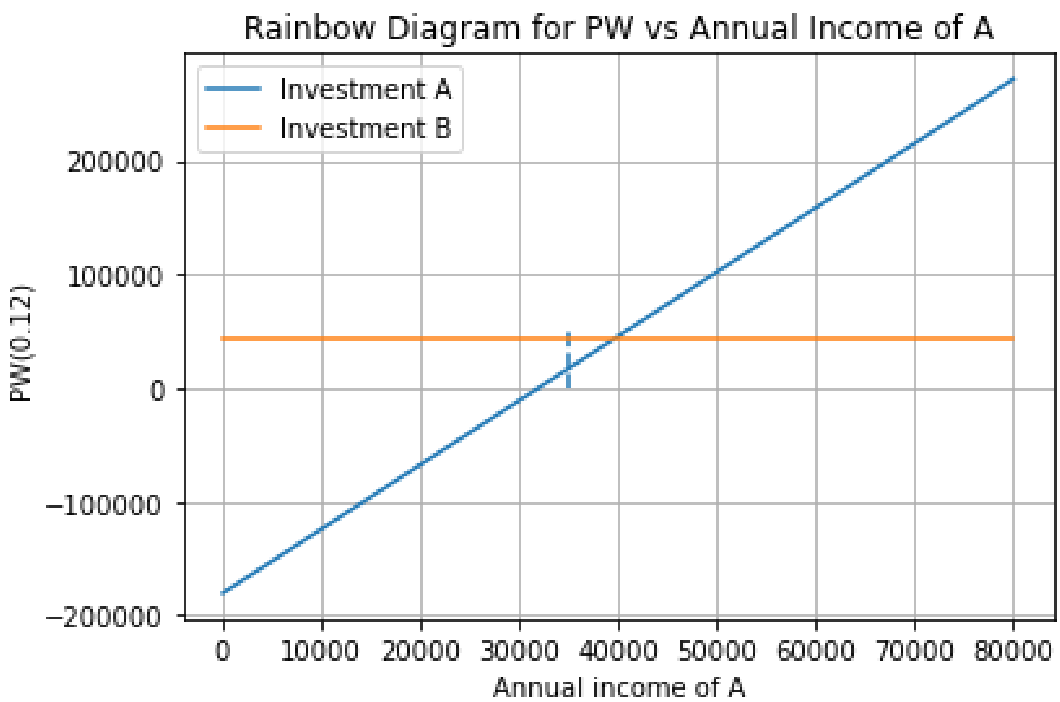



```
Reveral point = 143,173.35
Change in value required for decision reversal = -26,826.65
%-change = -15.78%
```

[8]:
```
""" Sensitivity analysis on Annual Income of Investment A """
print("\nSensitivity Analysis on Annual Income of Investment A")
def PW_A_R(R):
    return PW_A(Life, I_A, R, E_A, SV_A)

# Plot rainbow diagram
x = np.linspace(0, 80000, 101)
f3, ax3 = plt.subplots()
ax3.plot(x, PW_A_R(x), label='Investment A')
ax3.plot(x, [PW_B_base]*len(x), label='Investment B')
```

```
ax3.vlines(R_A, 0, 1.2*max(PW_A_R(R_A), PW_B_base),ls='--')
ax3.legend()
ax3.set_title("Rainbow Diagram for PW vs Annual Income of A")
ax3.set_xlabel("Annual income of A")
ax3.set_ylabel(f"PW({marr})")
ax3.grid()
plt.show()

# Find break point
guess = 40000
# Solve PW_A(x) = PW_B_base
bp = root(lambda x: PW_A_R(x)-PW_B_base, guess, tol=1e-10).x
print(f"\nReveral point = {bp[0]:,.2f}")
print(f"Change in value required for decision reversal = {bp[0]-R_A:,.2f}")
print(f"%-change = {100*(bp[0]-R_A)/R_A:.2f}%")
```

```
Sensitivity Analysis on Annual Income of Investment A
```

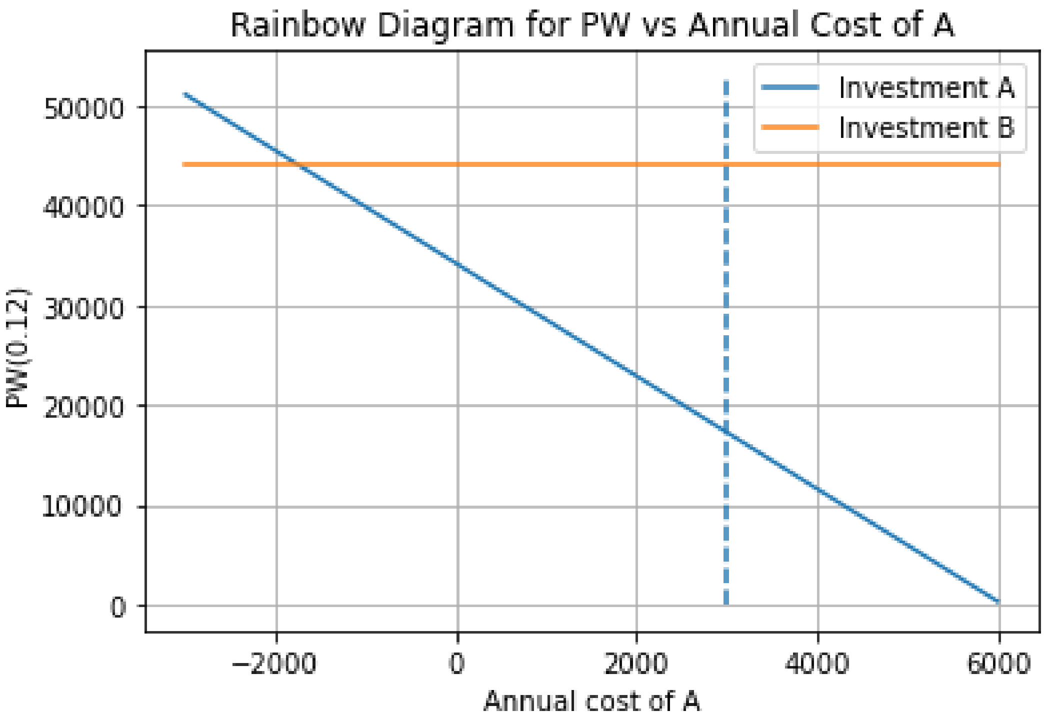



```
Reveral point = 39,747.89
Change in value required for decision reversal = 4,747.89
%-change = 13.57%
```

[9]:
```
""" Sensitivity analysis on Annual Cost of Investment A """
print("\nSensitivity Analysis on Annual Cost of Investment A")
def PW_A_E(x):
    return PW_A(Life, I_A, R_A, x, SV_A)
```

```
# Plot rainbow diagram
x = np.linspace(-3000, 6000, 101)
f4, ax4 = plt.subplots()
ax4.plot(x, PW_A_E(x), label='Investment A')
ax4.plot(x, [PW_B_base]*len(x), label='Investment B')
ax4.vlines(E_A, 0, 1.2*max(PW_A_E(E_A), PW_B_base), ls='--')
ax4.legend()
ax4.set_title("Rainbow Diagram for PW vs Annual Cost of A")
ax4.set_xlabel("Annual cost of A")
ax4.set_ylabel(f"PW({marr})")
ax4.grid()
plt.show()

# Find break point
guess = 2000
bp = root(lambda x: PW_A_E(x)-PW_B_base, guess, tol=1e-10).x
print(f"\nReveral point = {bp[0]:,.2f}")
print(f"Change in value required for decision reversal = {bp[0]-E_A:,.2f}")
print(f"%-change = {100*(bp[0]-E_A)/E_A:.2f}%")
```

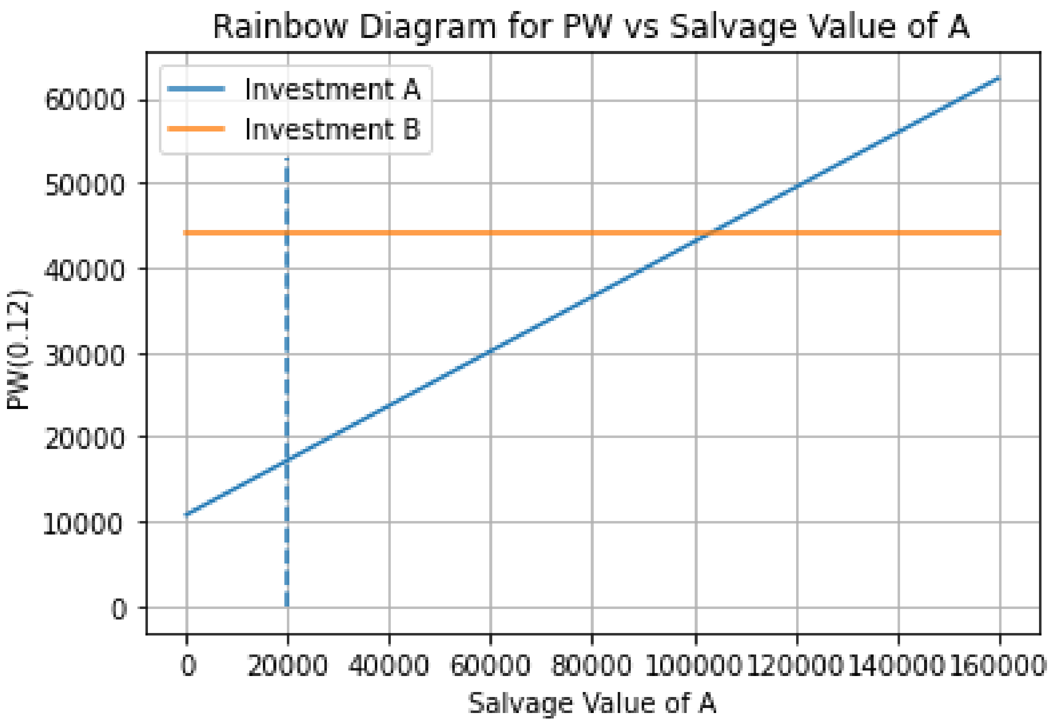
```
Sensitivity Analysis on Annual Cost of Investment A
```




```
Reveral point = -1,747.89
Change in value required for decision reversal = -4,747.89
%-change = -158.26%
```

[10]:
```python
""" Sensitivity analysis on Salvage Value of Investment A """
print("\nSensitivity Analysis on Salvage Value of Investment A")
def PW_A_SV(x):
    return PW_A(Life, I_A, R_A, E_A, x)

# Plot rainbow diagram
x = np.linspace(0, 160000, 101)
f5, ax5 = plt.subplots()
ax5.plot(x, PW_A_SV(x), label='Investment A')
ax5.plot(x, [PW_B_base]*len(x), label='Investment B')
ax5.vlines(SV_A, 0, 1.2*max(PW_A_SV(SV_A), PW_B_base), ls='--')
ax5.legend()
ax5.set_title("Rainbow Diagram for PW vs Salvage Value of A")
ax5.set_xlabel("Salvage Value of A")
ax5.set_ylabel(f"PW({marr})")
ax5.grid()
plt.show()

# Find break point
guess = 100000
bp = root(lambda x: PW_A_SV(x)-PW_B_base, guess, tol=1e-10).x
print(f"\nReveral point = {bp[0]:,.2f}")
print(f"Change in value required for decision reversal = {bp[0]-SV_A:,.2f}")
print(f"%-change = {100*(bp[0]-SV_A)/SV_A:.2f}%")
```

```
Sensitivity Analysis on Salvage Value of Investment A
```



```
Reveral point = 103,319.51
Change in value required for decision reversal = 83,319.51
%-change = 416.60%
```

[ ]:

8 Probabilitic Risk Analysis

8.1 Class Monte_Carlo_Simulation

[1]:
```
from EngFinancialPy import Monte_Carlo_Simulation
```

[2]:
```
print(Monte_Carlo_Simulation.__doc__)
```

```
 Perform Monte Carlo Simulation and Probabilsitic Risk Analysis
    Parameters:
      fixed_vars = dictionary of fixed variables name and value
      random_vars = dictionary of random variable name and stats objects
      ouput_functions = dictionary of output name and functions
    Methods:
      base_case: returns dictionary of output base case values
      run: run the simulation model
      show_inputs_values: show statistics and distributions of input values
      show_outputs_values: show statistics and distributions of outputs
      Prob_Analysis_DCF: perform probabilistic risk analysis on DCF outputs
      Prob_Analysis_rate: perform probabilistic risk analysis of rate outputs
```

[ ]:

8.2 Monte Carlo Simulation Example

Source: 6.5.3_monte_carlo_simulation.ipynb

[1]:
```
# 6.5.3_monte_carlo_simulation.ipynb
""" 6.5.3 Monte Carlo Simulation Example """
from EngFinancialPy import Monte_Carlo_Simulation
import numpy_financial as npf
from scipy import stats
```

[2]:
```
# Fixed input variables' name and value
fixed_vars = {'marr': 0.08, 'I': -150000 }
```

[3]:
```
# Random input variables' name and random variable objects
# See https://docs.scipy.org/doc/scipy/reference/stats.html for details
random_vars = {'R' : stats.norm( 70000, 4000),
               'E' : stats.norm(-43000, 2000),
               'SV': stats.uniform(1000, 3000-1000),
               'Life' : stats.randint(8, 12+1) }
```

[4]:
```python
# Define functions to compute output variable's values
# Arrange the arguments in the same order as above
def PW(marr, I, R, E, SV, Life):
    return I - npf.pv(marr, Life, R+E, SV)

def IRR(marr, I, R, E, SV, Life):
     return npf.rate(Life, R+E, I, SV)
```

[5]:
```python
# The output variable's name and functions for the simulation
output_functions = {'PW': PW, 'IRR': IRR }
```

[6]:
```python
# Create a simulation model instance with above data
sim_model = Monte_Carlo_Simulation(fixed_vars,random_vars,output_functions)
```

[7]:
```python
# Perform base case analysis when all variables are at their mean values
for name, value in sim_model.base_case().items():
    print(f"Base case value of {name} = {value:.4f}")
```

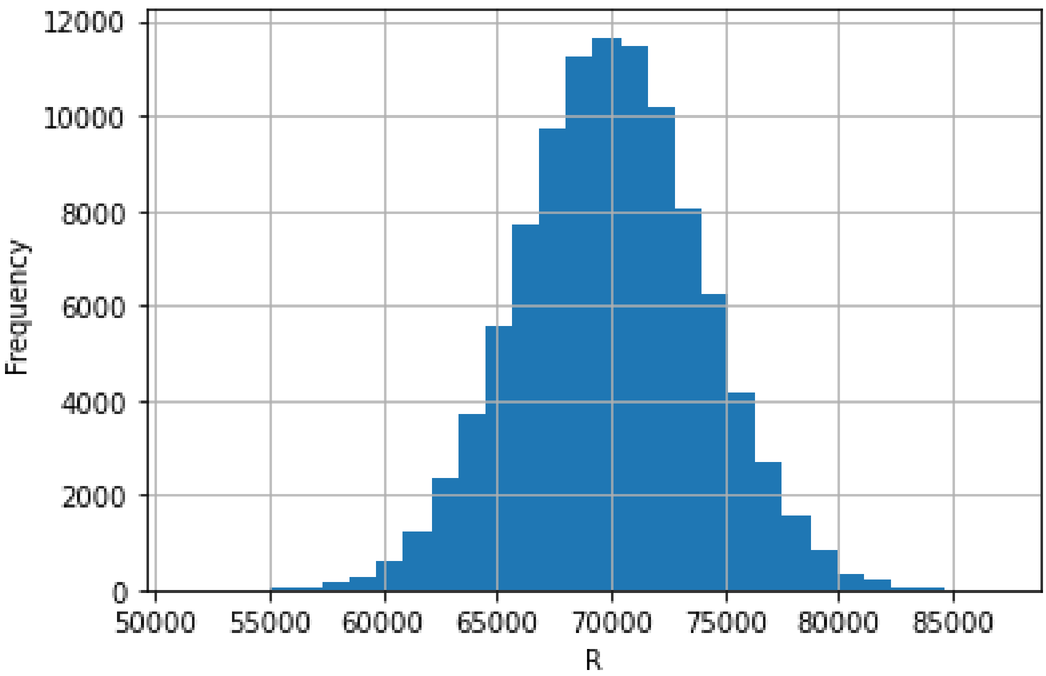
```
Base case value of PW = 32098.5847
Base case value of IRR = 0.1252
```

[8]:
```python
# Perform Monte Carlo Simulation.
status = sim_model.run(num_trials=100000)
print(status)
```

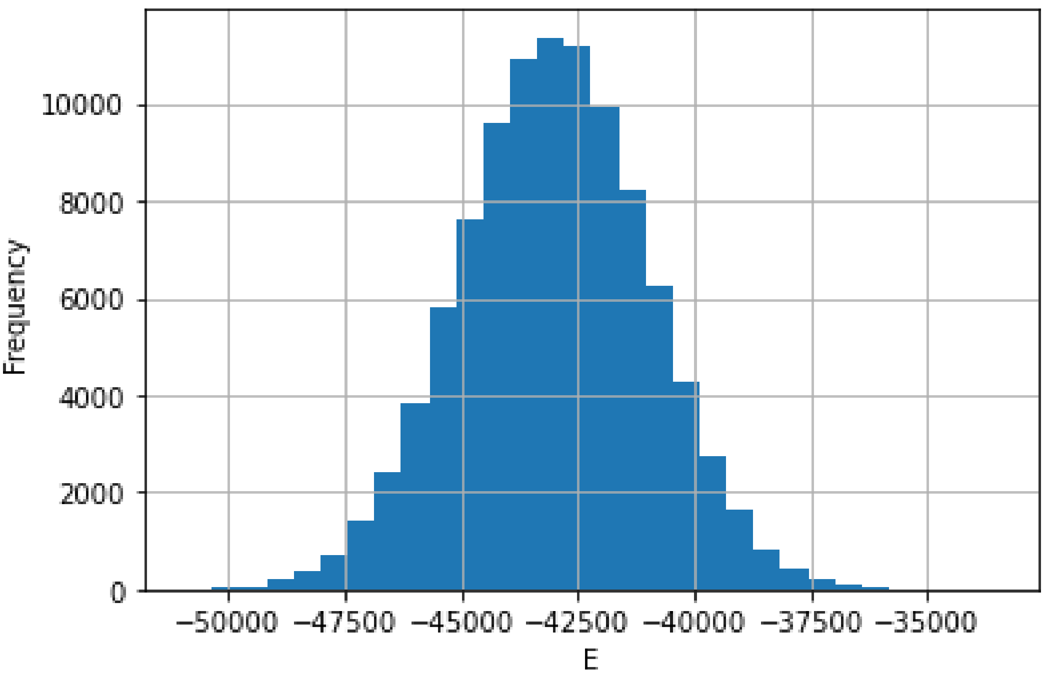
```
Simulation Completed
```

[9]:
```python
# Show input variables statistics and distribution
sim_model.show_inputs_values()
```

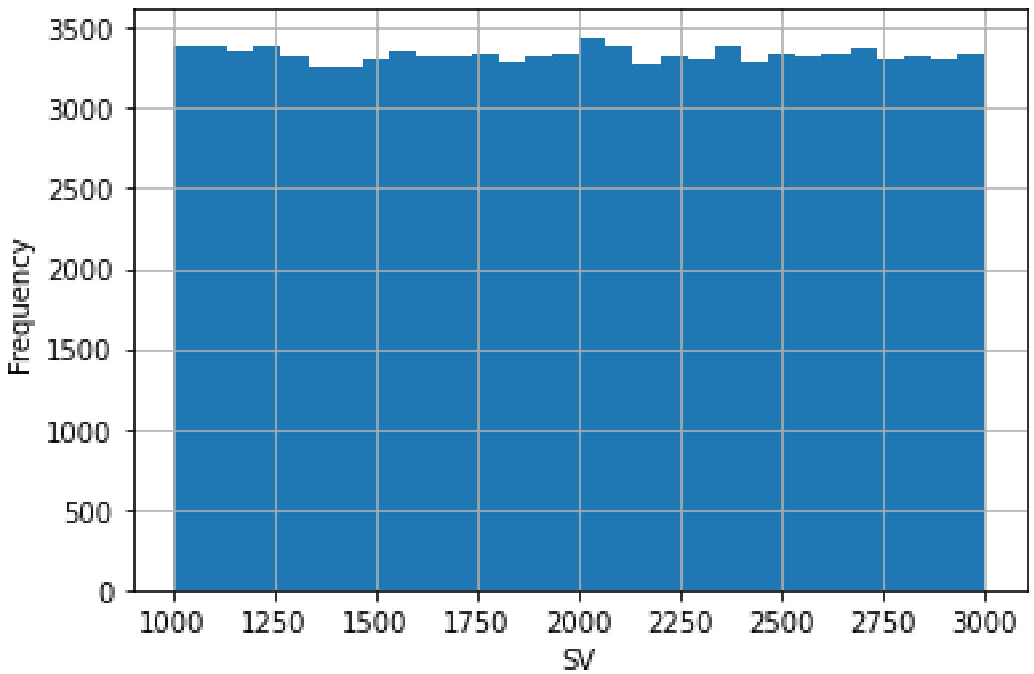
```
Input Variable R:
count    100000.00
mean      69997.66
std        3999.47
min       51403.01
25%       67302.89
50%       69983.50
75%       72680.58
max       87064.03
Name: R, dtype: float64

Histogram:
```



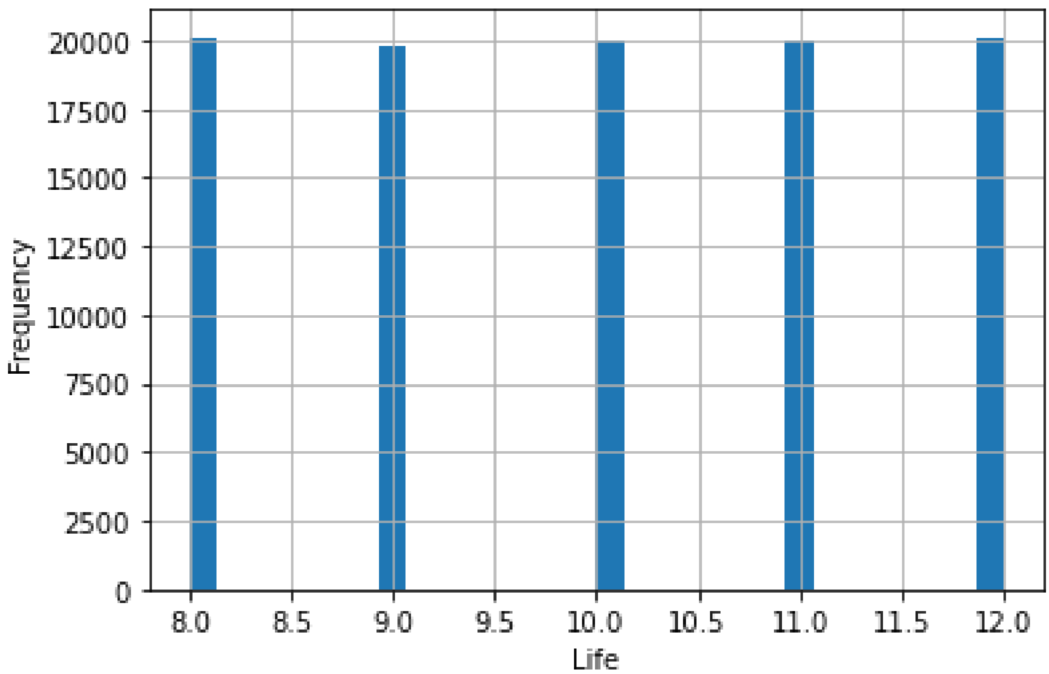

```
Input Variable E:
count     100000.00
mean      -43005.05
std         2003.10
min       -50907.06
25%       -44355.53
50%       -43002.39
75%       -41647.04
max       -33501.89
Name: E, dtype: float64

Histogram:
```




```
Input Variable SV:
count    100000.00
mean       1999.10
std         577.72
min        1000.03
25%        1498.79
50%        2001.04
75%        2499.40
max        2999.99
Name: SV, dtype: float64

Histogram:
```

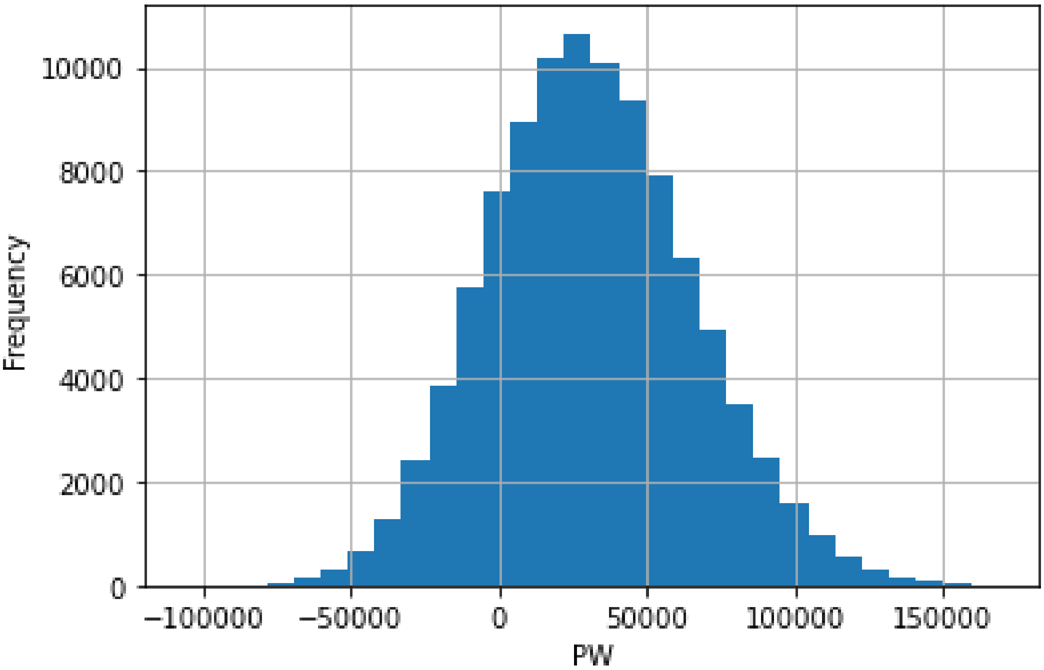


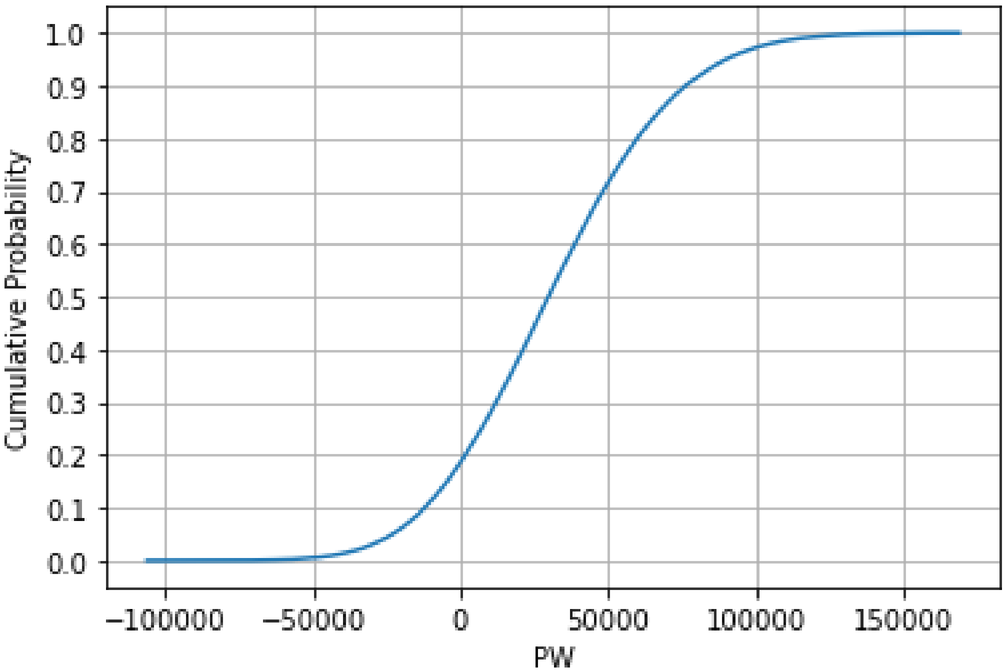

```
Input Variable Life:
count     100000.00
mean          10.00
std            1.42
min            8.00
25%            9.00
50%           10.00
75%           11.00
max           12.00
Name: Life, dtype: float64

Histogram:
```

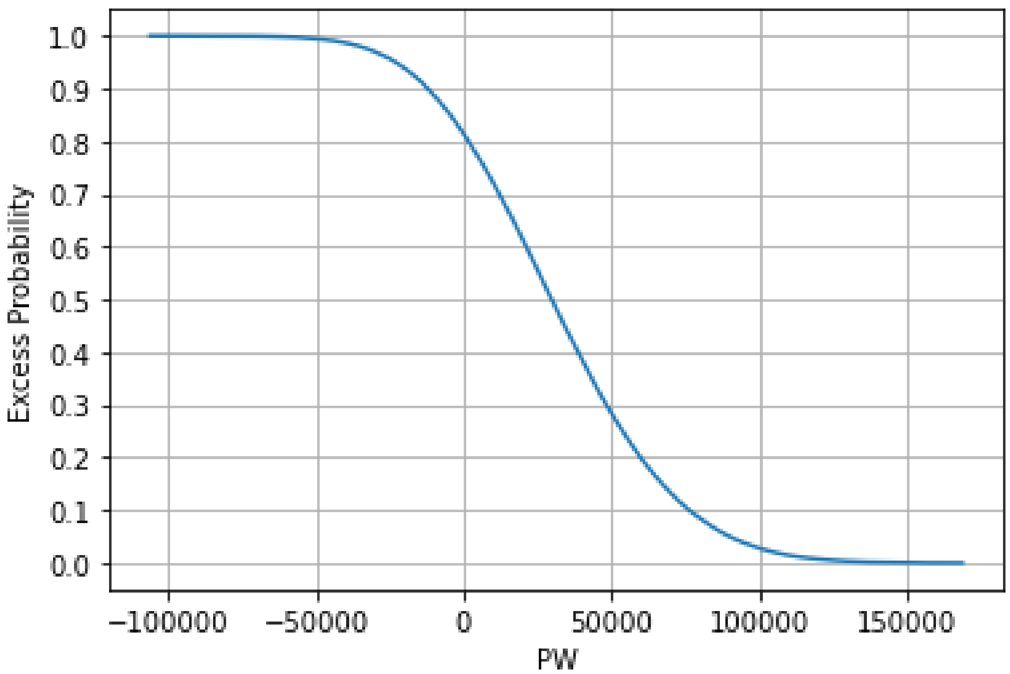


[10]:
```
# show output variables statistics and distribution
sim_model.show_outputs_values()
```

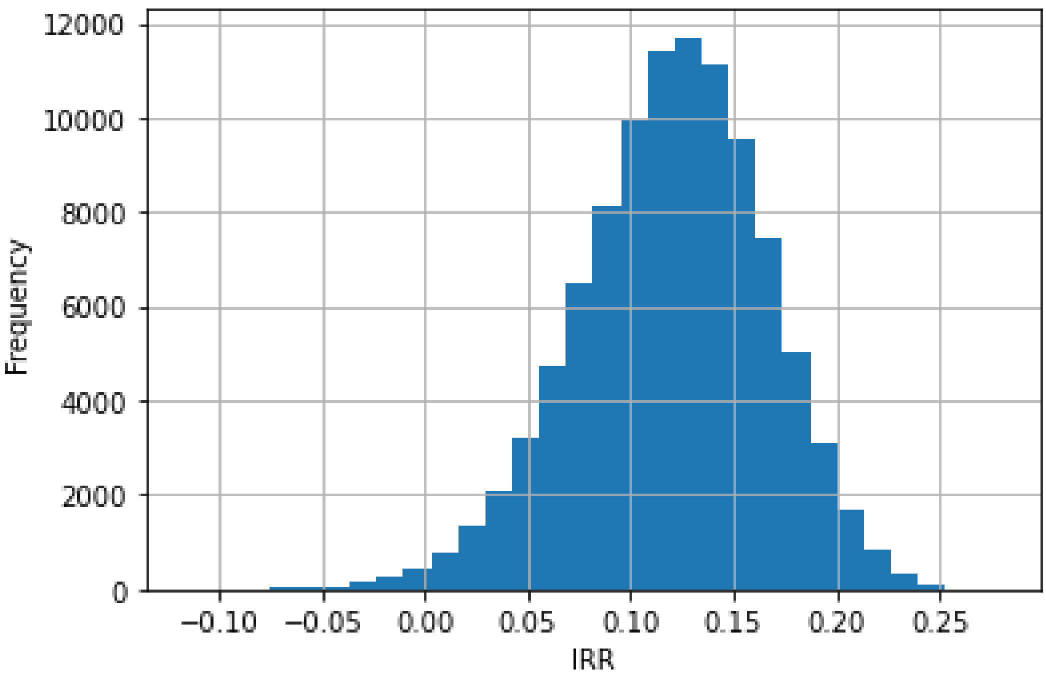
```
Output Variable PW:
count      99991.00
mean       31112.98
std        34409.06
min      -106150.23
25%         7136.87
50%        29770.09
75%        53739.07
max       168777.87
Name: PW, dtype: float64

Histogram:
```

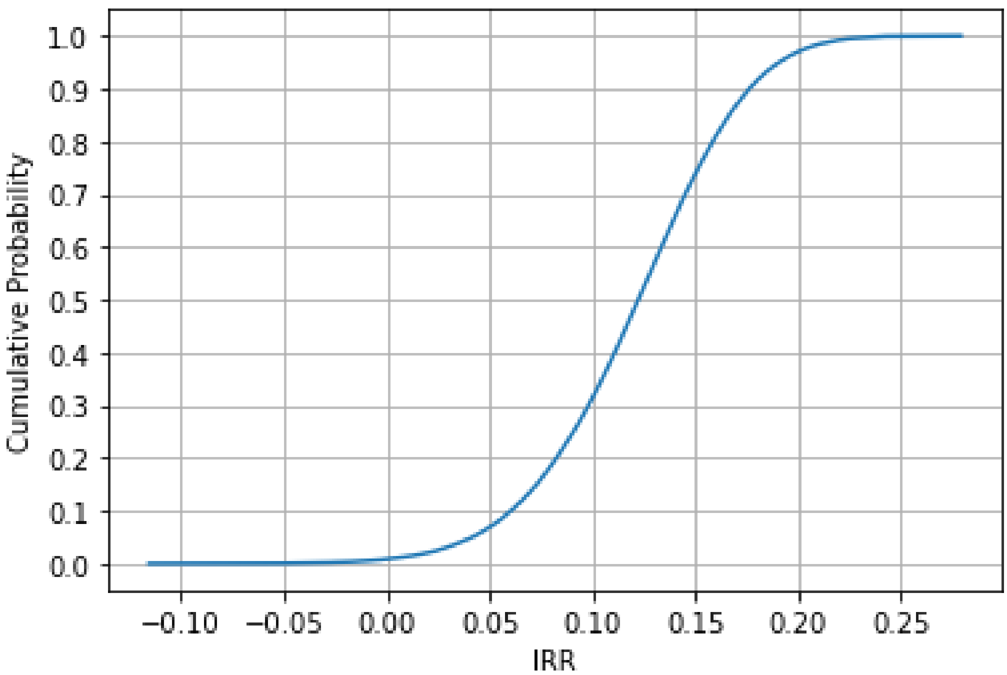

10000
8000
6000
4000
2000
0
Frequency
−100000
−50000
0
50000
100000
150000
PW

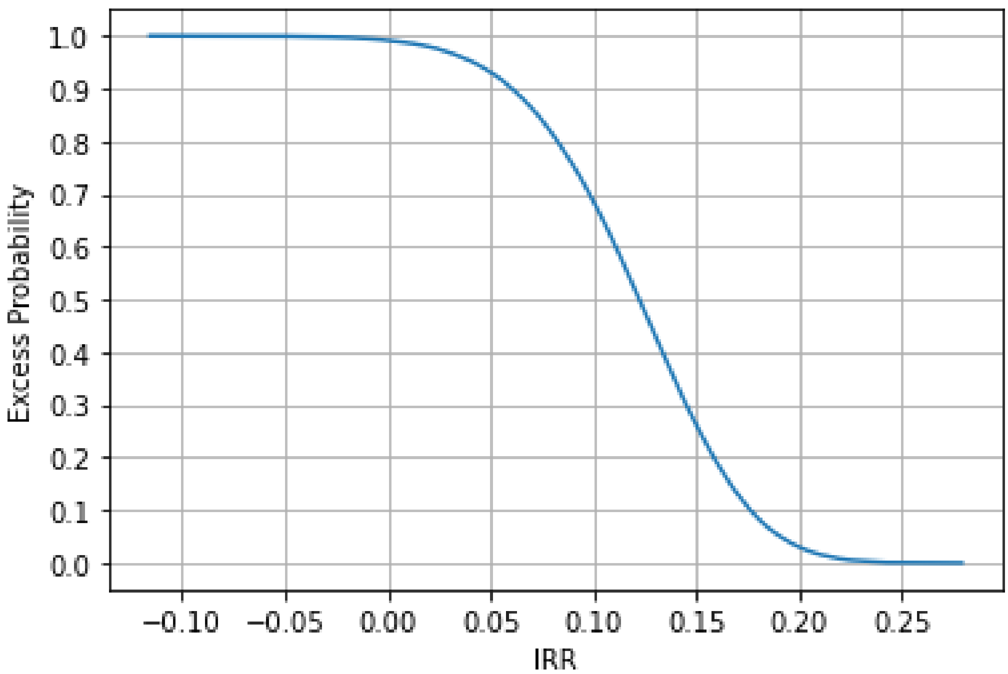

1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Cumulative Probability
−100000
−50000
0
50000
100000
150000
PW




```
Output Variable IRR:
count    99954.00
mean         0.12
std          0.05
min         -0.12
25%          0.09
50%          0.12
75%          0.15
max          0.28
Name: IRR, dtype: float64

Histogram:
```


12000
10000
8000
6000
4000
2000
0
Frequency
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
0.25
IRR


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Cumulative Probability
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
0.25
IRR




```
[11]: # Perform probabilistic risk analysis on output variable PW
sim_model.Prob_Analysis_DCF('PW',
            downsides=[-10000, -5000, 0],
            upsides=[20000, 40000, 60000, 80000, 100000])
```

```
Probabilistic Analysis on PW
  EV = 31,073.67
  SD = 34,312.21
Downside Risks:
  Pr(PW <= -10,000) = 11.27%
  Pr(PW <=  -5,000) = 14.63%
  Pr(PW <=       0) = 18.59%
Upside Potentials:
  Pr(PW >=  20,000) = 61.32%
  Pr(PW >=  40,000) = 38.61%
  Pr(PW >=  60,000) = 19.84%
  Pr(PW >=  80,000) = 8.39%
  Pr(PW >= 100,000) = 2.77%
Value-at-Risk:
  VaR(99%) =  43,093.17
  VaR(95%) =  23,050.16
  VaR(90%) =  12,178.78
```

```
[12]: # Perform probabilistic risk analysis on output variable IRR
marr = fixed_vars['marr']
sim_model.Prob_Analysis_rate('IRR', marr,
            downsides =[marr-0.02, marr-0.04],
            upsides=[0.10, 0.15, 0.20])
```

```
Probabilistic Analysis on IRR:
  EV = 12.00%
  SD = 4.55%
Downside Risks:
  Pr(IRR <=  8.0%) = 18.59%
  Pr(IRR <=  6.0%) = 9.86%
  Pr(IRR <=  4.0%) = 4.71%
Upside Potentials:
  Pr(IRR >= 10.0%) = 68.62%
  Pr(IRR >= 15.0%) = 26.27%
  Pr(IRR >= 20.0%) = 3.03%
```

[ ]:

9 After-Tax Cash Flows Analysis of Projects

9.1 Class ATCF_Analysis

[1]:
```
from EngFinancialPy import ATCF_Analysis
```

[2]:
```
print(ATCF_Analysis.__doc__)
```

```
 After=Tax Cash Flow Analysis Class
    Parameters:
        btcf = Unsorted list of before tax cash flows, as tuples as follows:
             (EoY,'C', Value) for capital cash flows
             (EoY,'D', Value) for depreciations
             (EoY,'T', Value) for taxable or tax-deductible cash flows
             (EoY,'S', MV_n, BV_n) for asset disposal cash flow

        btcf can take multiple capital cash flows and salvage values
        There should be at least one entry of any type for each year
        from 0 to N. Use zero values if needed for any year with
        no cash flows of any type.
     Methods:
        atcf : a list of year-by-year after-tax cash flows
        atcf_table(silence=False): Returns ATCF table (DataFrame)
                                   Don't print table if silence=True
        after_tax_NPV(marr): Compute after-tax NPV at marr
        after_tax_PW(marr):  Compute after-tax PW at marr
        after_tax_AW(marr):  Compute after-tax AW at marr
        after_tax_FW(marr):  Compute after-tax FW at marr
        after_tax_IRR(marr): Compute after-tax IRR
```

[ ]:

9.2 After-Tax Cash Analysis with 1-Year Capital Allowance

Source: 7.4.6_ATCF_analysis_1Year_CA.ipynb

[1]:
```python
# 7.4.6_ATCF_analysis_1Year_CA.ipynb
""" 7.4.6 After-tax cash flow analysis under 1-Year Capital Allowance """
from EngFinancialPy import ATCF_Analysis
```

[2]:
```python
InitCost = 100000
a_profit = 25000
MV6 = 10000
BV6 = 0
marr = 0.1
tax_rate = 0.17
```

[3]:
```python
# Create a list of BTCF
BTCF1 = [ (0,'C', -InitCost),
          (1,'D',  InitCost),
          (6,'S',  MV6, BV6),
          (1,'T',  a_profit),
          (2,'T',  a_profit),
          (3,'T',  a_profit),
          (4,'T',  a_profit),
          (5,'T',  a_profit),
          (6,'T',  a_profit) ]
```

[4]:
```python
# Create an ATCF_Analysis instance
OneY = ATCF_Analysis(BTCF1, tax_rate=tax_rate)
```

[5]:
```python
# Show the ATCF Table
OneY.atcf_table()
# Compute after-tax profitability measures
print(f"After-tax PW  = {OneY.after_tax_PW(marr):9,.2f}")
print(f"After-tax AW  = {OneY.after_tax_AW(marr):9,.2f}")
print(f"After-tax FW  = {OneY.after_tax_FW(marr):9,.2f}")
print(f"After-tax IRR = {OneY.after_tax_IRR()*100:9.2f}%")
```

```
   After-Tax Cash Flow Analysis Table
   EoY    BTCF Depreciation Taxable Income IT Cash Flow        ATCF
0    0 -100000                                          -100,000.00
1    1   25000       100000         -75000    12,750.00   37,750.00
2    2   25000            0          25000    -4,250.00   20,750.00
3    3   25000            0          25000    -4,250.00   20,750.00
4    4   25000            0          25000    -4,250.00   20,750.00
5    5   25000            0          25000    -4,250.00   20,750.00
7    6   25000            0          25000    -4,250.00   20,750.00
6    6   10000                   10,000.00    -1,700.00    8,300.00
After-tax PW  = 10,511.34
After-tax AW  =  2,413.48
After-tax FW  = 18,621.48
After-tax IRR =     13.76%
```

[6]:
```python
# You can also directly print the ATCF Table
ATCF_Analysis(BTCF1, tax_rate=tax_rate).atcf_table()
```

```
   After-Tax Cash Flow Analysis Table
```

```
   EoY    BTCF Depreciation Taxable Income IT Cash Flow         ATCF
0    0 -100000                                             -100,000.00
1    1   25000       100000          -75000    12,750.00   37,750.00
2    2   25000            0           25000    -4,250.00   20,750.00
3    3   25000            0           25000    -4,250.00   20,750.00
4    4   25000            0           25000    -4,250.00   20,750.00
5    5   25000            0           25000    -4,250.00   20,750.00
7    6   25000            0           25000    -4,250.00   20,750.00
6    6   10000                    10,000.00    -1,700.00    8,300.00
```

[6]:
```
   EoY    BTCF Depreciation Taxable Income IT Cash Flow         ATCF
0    0 -100000                                             -100,000.00
1    1   25000       100000          -75000    12,750.00   37,750.00
2    2   25000            0           25000    -4,250.00   20,750.00
3    3   25000            0           25000    -4,250.00   20,750.00
4    4   25000            0           25000    -4,250.00   20,750.00
5    5   25000            0           25000    -4,250.00   20,750.00
7    6   25000            0           25000    -4,250.00   20,750.00
6    6   10000                    10,000.00    -1,700.00    8,300.00
```

[7]:
```
# You can also compute the after-tax PW directly and check its feasibility
if ATCF_Analysis(BTCF1, tax_rate=tax_rate).after_tax_PW(marr) >= 0:
    print("Project is feasible")
else:
    print("Project is not feasible")
```

```
Project is feasible
```

[ ]:
```

```

9.3 After-Tax Cash Analysis with 3-Year Capital Allowance

Source: 7.4.6_ATCF_analysis_3Year_CA.ipynb

[1]:
```
# 7.4.6_ATCF_analysis_3Year_CA.ipynb
""" 7.4.6 After-tax cash flow analysis under 3-Year Capital Allowance """
from EngFinancialPy import ATCF_Analysis
```

[2]:
```
InitCost = 100000
a_profit = 25000
MV6 = 10000
BV6 = 0
marr = 0.1
tax_rate = 0.17
```

[3]:
```
# Create a list of BTCF
BTCF3 = [ (0,'C', -InitCost),
          (1,'D',  InitCost/3),
          (2,'D',  InitCost/3),
          (3,'D',  InitCost/3),
          (6,'S',  MV6, BV6),
          (1,'T',  a_profit),
```

```
            (2,'T',  a_profit),
            (3,'T',  a_profit),
            (4,'T',  a_profit),
            (5,'T',  a_profit),
            (6,'T',  a_profit) ]
```

[4]:
```
# Create an ATCF_Analysis instance
ThreeY = ATCF_Analysis(BTCF3, tax_rate=tax_rate)
```

[5]:
```
# Show the ATCF Table
ThreeY.atcf_table()
# Compute after-tax profitability measures
print(f"After-tax PW  = {ThreeY.after_tax_PW(marr):9,.2f}")
print(f"After-tax AW  = {ThreeY.after_tax_AW(marr):9,.2f}")
print(f"After-tax FW  = {ThreeY.after_tax_FW(marr):9,.2f}")
print(f"After-tax IRR = {ThreeY.after_tax_IRR()*100:9.2f}%")
```

```
   After-Tax Cash Flow Analysis Table
   EoY    BTCF Depreciation Taxable Income IT Cash Flow        ATCF
0    0 -100000                                        -100,000.00
1    1   25000    33,333.33      -8,333.33     1,416.67   26,416.67
2    2   25000    33,333.33      -8,333.33     1,416.67   26,416.67
3    3   25000    33,333.33      -8,333.33     1,416.67   26,416.67
4    4   25000         0.00      25,000.00    -4,250.00   20,750.00
5    5   25000         0.00      25,000.00    -4,250.00   20,750.00
7    6   25000         0.00      25,000.00    -4,250.00   20,750.00
6    6   10000                   10,000.00    -1,700.00    8,300.00
After-tax PW  =  9,148.95
After-tax AW  =  2,100.67
After-tax FW  = 16,207.93
After-tax IRR =     13.12%
```

[ ]:
```

```

10 Replacement Analysis

10.1 Class Asset

[1]:
```
from EngFinancialPy import Asset
```

[2]:
```
print(Asset.__doc__)
```

```
 Asset class for capital asset economic replacement analysis
    Parameters:
      MV0 = current market value or initial cost
      MV = list of market values at EoY k for k = 1 to N
      E = list of annual expense in year k for k = 1 to N
      marr = marr
    Methods:
      useful_life:  Useful (remaining) life of asset
      EPC: Compute Equivalent Present Cost for year k, k = 1 to N
      TC:  Compute the total marginal costs for year k, k = 1 to N
```

```
EUAC_conventional(self): Compute the EUAC for year k, k = 1 to N
                         using Conventional method
EUAC:  Compute and the EUAC for year k, k = 1 to N using TC_k method
econ_life_euac: Compute ( Economic Service Life, Min EUAC )
TC_montotonic: True if the TC values are montonically non-decreasing
```

[ ]:

10.2 Function pprint_list

[1]:
```
from EngFinancialPy import pprint_list
```

[2]:
```
print(pprint_list.__doc__)
```

```
Pretty format print a List of numbers
```

[ ]:

10.3 Economic Service Life of New Assets

10.3.1 Machine A

Source: 8.3.5_economic_service_life_new_asset_machine_A.ipynb

[1]:
```
# 8.3.5_economic_service_life_new_asset_machine_A.ipynb
""" 8.3.5: Economic service life of new Machine A """
from EngFinancialPy import Asset, pprint_list
```

[2]:
```
# New Machine A data
InitCost = 13000
MV = [9000, 8000, 6000, 2000, 0 ]
E =  [2500, 2700, 3000, 3500, 4500]
marr = 0.1
```

[3]:
```
# Create a new Asset with age = 0
new_asset = Asset(InitCost, MV, E, marr, age=0, name="Machine A")
```

[4]:
```
# Compute all relavant outputs
print(new_asset.name)
pprint_list("EPC", new_asset.EPC())
```

```
Machine A
EPC = 7,090.91   10,892.56   15,250.19   20,782.60   24,942.77
```

[5]:
```
# Using Conventional EUAC formula
print("Using Conventional EUAC formula:")
pprint_list("EUAC", new_asset.EUAC_conventional())
```

```
Using Conventional EUAC formula:
EUAC = 7,800.00   6,276.19   6,132.33   6,556.30   6,579.84
```

[6]:
```
# Get TC values
pprint_list("TC", new_asset.TC())
```

```
TC = 7,800.00    4,600.00    5,800.00    8,100.00    6,700.00
```

[7]:
```
# Using TC to compute EUAC
pprint_list("EUAC", new_asset.EUAC())
```

```
EUAC = 7,800.00    6,276.19    6,132.33    6,556.30    6,579.84
```

[8]:
```
# Find economic service life and min EUAC
econ_life, euac_star = new_asset.econ_life_euac()
print(f"Economic Service Life = {econ_life} yrs at EUAC*={euac_star:,.2f}")
```

```
Economic Service Life = 3 yrs at EUAC*=6,132.33
```

[ ]:
```

```

10.3.2 New Forklift Truck

Source: 8.3.5_economic_service_life_new_forklift_truck.ipynb

[1]:
```
# 8.3.5_economic_service_life_new_forklift_truck.ipynb
""" 8.3.5 Econcomic service life of New forklift truck """
from EngFinancialPy import Asset, pprint_list
```

[2]:
```
# New Forklift Truck data
InitCost = 20000
MV = [15000, 11250, 8500, 6500,  4750 ]
E  = [ 2000,  3000, 4620, 8000, 12000 ]
marr = 0.1
```

[3]:
```
# Create a new asset with age = 0
new_forklift = Asset(InitCost, MV, E, marr, age=0, name="New forklift truck")
```

[4]:
```
# Compute all relavant outputs
print(new_forklift.name)
pprint_list("EPC", new_forklift.EPC())
pprint_list("TC", new_forklift.TC())
pprint_list("EUAC", new_forklift.EUAC())
econ_life, euac_star = new_forklift.econ_life_euac()
print(f"Economic Service Life = {econ_life} yrs at EUAC*= {euac_star:,.2f}")
```

```
New forklift truck
EPC = 8,181.82    15,000.00    21,382.42    28,793.12    37,734.38
TC = 9,000.00    8,250.00    8,495.00    10,850.00    14,400.00
EUAC = 9,000.00    8,642.86    8,598.19    9,083.39    9,954.23
Economic Service Life = 3 yrs at EUAC*= 8,598.19
```

[ ]:
```

```

10.4 Replacement of Defenders under Infinite Planning Horizon

10.4.1 When the Defender's TC are not monotoically non-decreasing

Source: 8.4.3_replacement_analysis_infinite_horizon_def_TC_not_non_decreasing.ipynb

[1]:
```
# 8.4.3_replacement_analysis_infinite_horizon_def_TC_not_non_decreasing.ipynb
""" 8.4.3 Optimal Replacement under Infinite Planning Horizon when
    Defender TC values are not monotoically non-decreasing """
import numpy_financial as npf
import matplotlib.pyplot as plt
from EngFinancialPy import Asset, pprint_list
```

[2]:
```
# Defender data
MV0 = 16000
MV = [10600, 7800, 5600, 3600, 2000, 1200 ]
E  = [ 3000, 4200, 5400, 6800, 8400, 9800 ]
marr = 0.1
```

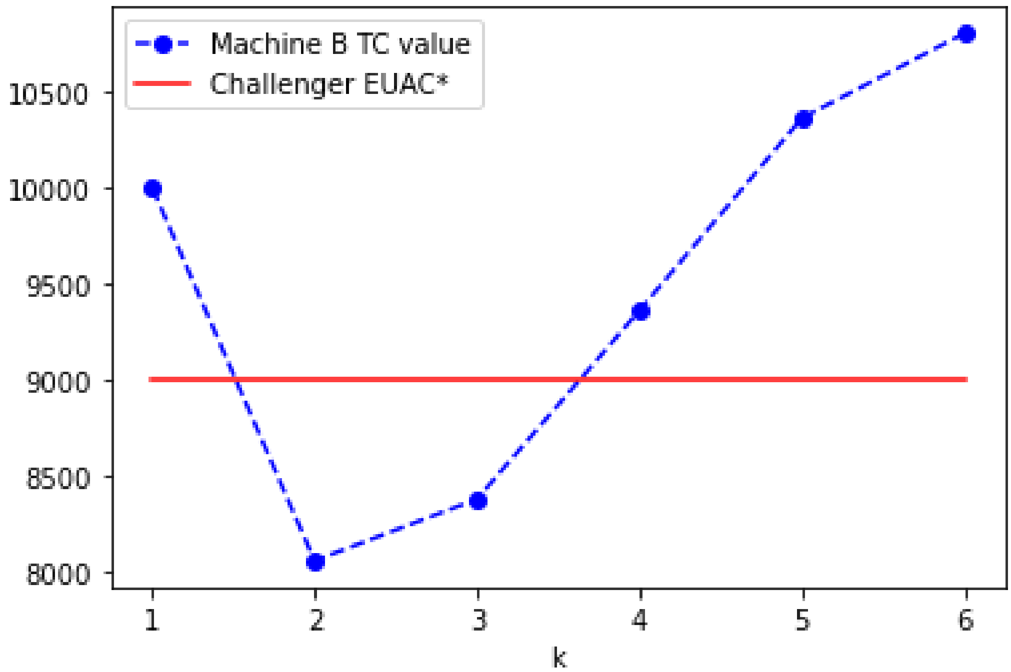
[3]:
```
# Create defdende instance
defender = Asset(MV0, MV, E, marr, age=5, name="Machine B")
```

[4]:
```
# The best challenger's EUAC
challenger_euac = 9000
```

[5]:
```
# Check defender's TC values
TC_def = defender.TC()
pprint_list("TC", TC_def)
```

```
TC = 10,000.00    8,060.00    8,380.00    9,360.00    10,360.00    10,800.00
```

[6]:
```
# Plot the defender's TC values
defender.plot_TC(challenger_euac)
```



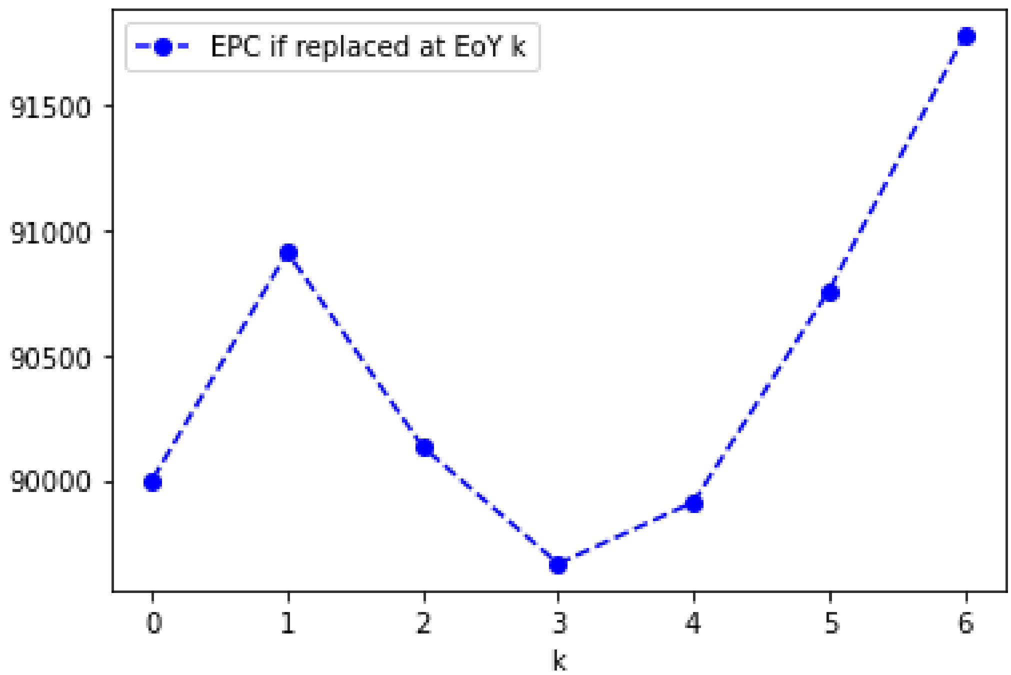
[7]:
```
# Check if defender's TC values are monotoically non-decreasing
if defender.TC_montotonic():
    print("The Defender's TC values are monotonically non-decreasing")
else:
    print("The Defender's TC values are not monotonically non-decreasing")
```

```
The Defender's TC values are not monotonically non-decreasing
```

[8]:
```
# Compute the EPC if defender is replaced by repeatable challenger
# after k years, for k = 0 to N.
life = defender.useful_life()
EPC = [npf.npv(marr, [0]+TC_def[:k]) + challenger_euac/(marr*(1+marr)**k)
            for k in range(0, life+1) ]
pprint_list("EPC", EPC)
```

```
EPC = 90,000.00   90,909.09   90,132.23   89,666.42   89,912.30   90,756.75
91,772.81
```

[9]:
```
# Plot the replacement plans' EPC values
fig, ax = plt.subplots()
ax.plot(range(life+1), EPC,'bo',ls='--',label='EPC if replaced at EoY k')
ax.set_xticks(range(life+1))
ax.set_xlabel("k")
ax.legend()
plt.show()
```



```
[10]: # Determine the optimal replacement time.
epc_star = min(EPC)
kstar = EPC.index(epc_star)
print(f"Replace the defender with repeatable challenger at EoY {kstar}")
print(f"Optimal EPC under opportunity cost approach = {epc_star:,.2f}")
euac_star_CF = (epc_star - MV0)*marr
print(f"Optimal EUAC under cash flow approach = {euac_star_CF:,.2f}")
```

```
Replace the defender with repeatable challenger at EoY 3
Optimal EPC under opportunity cost approach = 89,666.42
Optimal EUAC under cash flow approach = 7,366.64
```

```
[ ]:
```

10.4.2 When the Defender's TC are monotoically non-decreasing

Source: 8.4.4_replacement_analysis_infinite_horizon_def_TC_non_decreasing.ipynb

```
[1]: # 8.4.4_replacement_analysis_infinite_horizon_def_TC_non_decreasing.ipynb
""" 8.4.4 Optimal Replacement under Infinite Planning Horizon when
    Defender TC values are monotonically non-decreasing """
import numpy_financial as npf
import matplotlib.pyplot as plt
from EngFinancialPy import Asset, pprint_list
```

```
[2]: # Defender forklift truck
MV0 = 5000
MV = [4000, 3000, 2000, 1000]
E  = [5500, 6600, 7800, 8800 ]
marr = 0.1
```

```
[3]: def_forklift = Asset(MV0, MV, E, marr, age=2, name="Defender forklift truck")
```

```
[4]: # Get defender's TC values
TC_def = def_forklift.TC()
pprint_list("TC", TC_def)
```

```
TC = 7,000.00    8,000.00    9,100.00    10,000.00
```

```
[5]:  def_forklift = Asset(MV0, MV, E, marr, age=2, name="Defender forklift truck")
```

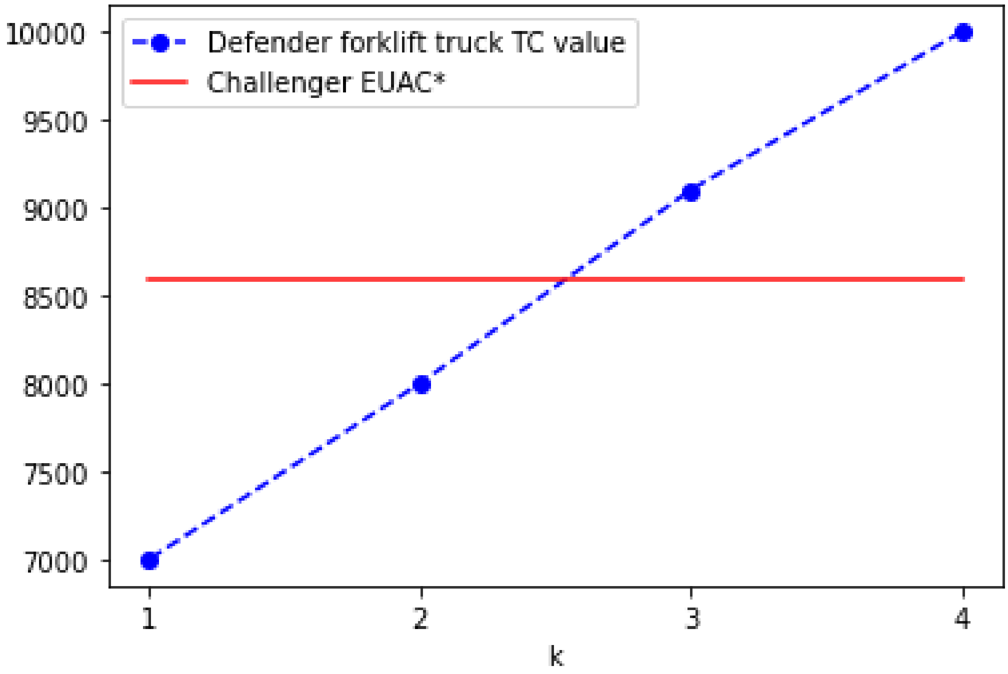
```
[6]: # Best challenger EUAC under repeatability assumption.
MV0_new = 20000
MV_new = [15000, 11250, 8500, 6500,  4750 ]
E_new  = [ 2000,  3000, 4620, 8000, 12000 ]
```

```
[7]: new_forklift = Asset(MV0_new, MV_new, E_new, marr, age=0,
                          name="New forklift truck")
```

```
[8]: # Best challenger EUAC under repeatability assumption
challenger_econ_life, challenger_euac = new_forklift.econ_life_euac()
print(challenger_euac)
```

8598.187311178239

[9]:
```
# Plot the defender's TC values
def_forklift.plot_TC(challenger_euac)
if def_forklift.TC_montotonic():
    print("The Defender's TC values are montonically non-decreasing")
else:
    print("The Defender's TC values are not montonically non-decreasing")
```



The Defender's TC values are montonically non-decreasing

[10]:
```
# Determine the optimal replacement time.
life = def_forklift.useful_life()
keep_years = [ t for t in range(0,life) if TC_def[t]<=challenger_euac ]
kstar = len(keep_years)
print(f"Replace the defender with repeatable challenger at EoY {kstar}")
```

Replace the defender with repeatable challenger at EoY 2

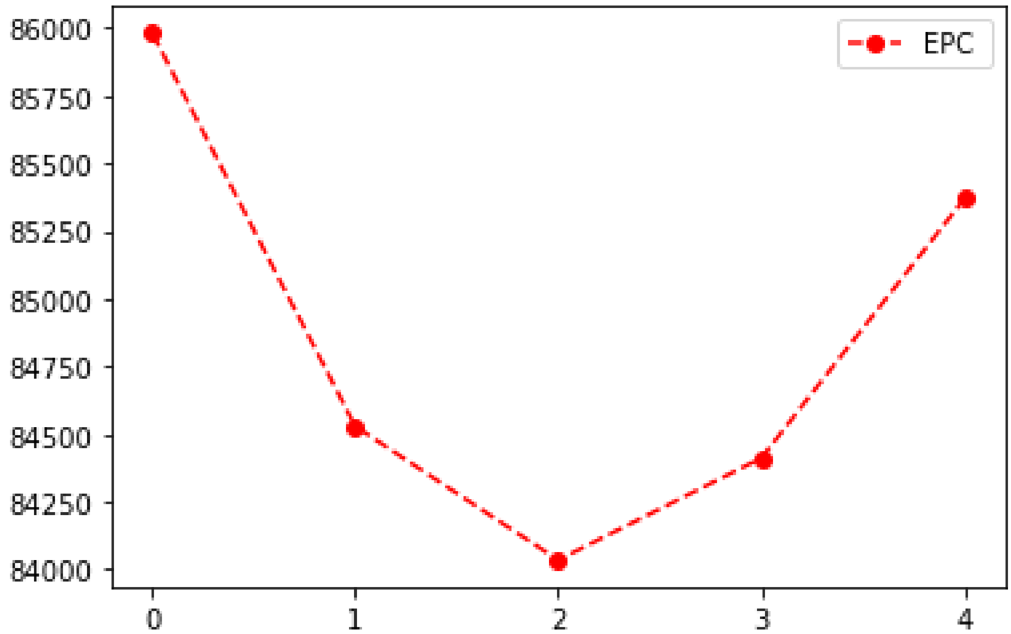
[11]:
```
# Determine the optimal EPC and EUAC
EPC_star = npf.npv(marr, [0]+TC_def[:kstar]) \
                   + challenger_euac/(marr*(1+marr)**kstar)
print(f"Optimal EPC under opportunity cost approach = {EPC_star:,.2f}")
EUAC_star_CF = (EPC_star - MV0)*marr
print(f"Optimal EUAC under cash flow approach = {EUAC_star_CF:,.2f}")
```

Optimal EPC under opportunity cost approach = 84,034.61
Optimal EUAC under cash flow approach = 7,903.46

[12]:
```python
""" Verify Solutions by minimizing Year-by-Year EPC(k)"""
# Compute the EPC if defender is replaced by repeatable challenger
# after k years, for k = 0 to N.
print("Verify Solution by Minimizing Year-by-Year EPC")
EPC = [npf.npv(marr, [0]+TC_def[:k]) + challenger_euac/(marr*(1+marr)**k)
       for k in range(0, life+1) ]
pprint_list("EPC", EPC)
```

```
Verify Solution by Minimizing Year-by-Year EPC
EPC = 85,981.87   84,528.98   84,034.61   84,411.63   85,369.08
```

[13]:
```python
# Plot the replacement plans' EPC values
fig, ax = plt.subplots()
ax.plot(range(life+1), EPC, 'ro', ls='--' , label='EPC ')
ax.set_xticks(range(life+1))
ax.legend()
plt.show()
```



[14]:
```python
# Determine the optimal replacement time.
epc_star = min(EPC)
kstar = EPC.index(epc_star)
print(f"Replace the defender with repeatable challenger at EoY {kstar}")
print(f"Optimal EPC under opportunity cost approach = {epc_star:,.2f}")

euac_star_CF = (epc_star - MV0)*marr
print(f"Optimal EUAC under cash flow approach = {euac_star_CF:,.2f}")
```

```
Replace the defender with repeatable challenger at EoY 2
Optimal EPC under opportunity cost approach = 84,034.61
Optimal EUAC under cash flow approach = 7,903.46
```

[ ]:

```

```

10.5 Replacement of Defenders under Finite Planning Horizon

10.5.1 Exhaustive Search Approach

Source: 8.5.1_replacement_analysis_finite_horizon.ipynb

[1]:

```
# 8.5.1_replacement_analysis_finite_horizon.ipynb
""" 8.5.1 Opitmal Replacement Planning under finite study period """
import numpy_financial as npf
from EngFinancialPy import Asset, pprint_list
```

[2]:

```
marr = 0.1
```

[3]:

```
# Defender data
MV0_d = 5000
MV_d = [4000, 3000, 2000, 1000 ]
E_d = [5500, 6600, 7800, 8800 ]
defender = Asset(MV0_d, MV_d, E_d, marr, name="Defender")
```

[4]:

```
# Challenger data
MV0_c = 20000
MV_c = [15000, 11250, 8500, 6500,  4750 ]
E_c = [ 2000,  3000, 4620, 8000, 12000 ]
challenger = Asset(MV0_c, MV_c, E_c, marr, age=0, name="Challenger")
```

[5]:

```
# Check defender's useful life TC values
life_d = defender.useful_life()
print(f"Defender remaining life = {life_d} years")
TC_d = defender.TC()
pprint_list("Defender TC", TC_d)
```

```
Defender remaining life = 4 years
Defender TC = 7,000.00   8,000.00   9,100.00   10,000.00
```

[6]:

```
# Check challenger's useful life and TC values
life_c = challenger.useful_life()
print(f"Challenger useful life = {life_c} years")
TC_c = challenger.TC()
pprint_list("Challenger TC", TC_c)
```

```
Challenger useful life = 5 years
Challenger TC = 9,000.00   8,250.00   8,495.00   10,850.00   14,400.00
```

[7]:

```
# Perform Replacement Analysis under finite planning horizon
print("\nReplacement Analysis under finitie planning horizon")
N = 6  # study period
print("Study period = 6 years, assume challenger is repeatable")
# Generate feasible replacement plans
plans = [(k1,k2,k3) for k1 in range(life_d+1) for k2 in range(life_c+1)
             for k3 in range(life_c+1) if (k1+k2+k3==N) and (k2>0 or k3==0)]
print(f"Number of feasible plans = {len(plans)}")
```

```
Replacement Analysis under finitie planning horizon
Study period = 6 years, assume challenger is repeatable
Number of feasible plans = 19
```

[8]:
```python
# Compute the Equivalent Present Cost of each plan
EPC_dt = {}
TC_CF_dt = {}
for k1,k2,k3 in plans:
    TC_CF = TC_d[0:k1] + TC_c[0:k2] + TC_c[0:k3]
    EPC = npf.npv(marr, [0]+TC_CF)
    EPC_dt[(k1,k2,k3)] = EPC
    TC_CF_dt[(k1,k2,k3)] = TC_CF

print("Plan, TC Cash Flows, EPC Table:")
for i, p in enumerate(plans):
    print(f"{i+1:3d}. {p}:", "".join(f"{x:7,.0f}" for x in TC_CF_dt[p]), f"␣
↪={EPC_dt[p]:10,.2f}")
```

```
Plan, TC Cash Flows, EPC Table:
  1. (0, 1, 5):   9,000  9,000  8,250  8,495 10,850 14,400  = 42,485.80
  2. (0, 2, 4):   9,000  8,250  9,000  8,250  8,495 10,850  = 38,795.96
  3. (0, 3, 3):   9,000  8,250  8,495  9,000  8,250  8,495  = 37,447.35
  4. (0, 4, 2):   9,000  8,250  8,495 10,850  9,000  8,250  = 39,038.32
  5. (0, 5, 1):   9,000  8,250  8,495 10,850 14,400  9,000  = 42,814.65
  6. (1, 1, 4):   7,000  9,000  9,000  8,250  8,495 10,850  = 37,597.62
  7. (1, 2, 3):   7,000  9,000  8,250  9,000  8,250  8,495  = 36,064.93
  8. (1, 3, 2):   7,000  9,000  8,250  8,495  9,000  8,250  = 36,047.40
  9. (1, 4, 1):   7,000  9,000  8,250  8,495 10,850  9,000  = 37,619.46
 10. (1, 5, 0):   7,000  9,000  8,250  8,495 10,850 14,400  = 40,667.62
 11. (2, 1, 3):   7,000  8,000  9,000  9,000  8,250  8,495  = 35,801.97
 12. (2, 2, 2):   7,000  8,000  9,000  8,250  9,000  8,250  = 35,617.10
 13. (2, 3, 1):   7,000  8,000  9,000  8,250  8,495  9,000  = 35,726.89
 14. (2, 4, 0):   7,000  8,000  9,000  8,250  8,495 10,850  = 36,771.17
 15. (3, 1, 2):   7,000  8,000  9,100  9,000  9,000  8,250  = 36,204.49
 16. (3, 2, 1):   7,000  8,000  9,100  9,000  8,250  9,000  = 36,162.16
 17. (3, 3, 0):   7,000  8,000  9,100  9,000  8,250  8,495  = 35,877.10
 18. (4, 1, 1):   7,000  8,000  9,100 10,000  9,000  9,000  = 37,310.86
 19. (4, 2, 0):   7,000  8,000  9,100 10,000  9,000  8,250  = 36,887.51
```

[9]:
```python
# Find optimal replacement plan
best_plan, min_EPC = min(EPC_dt.items(), key=lambda x: x[1])
print(f"\nOptimal plan = {best_plan}")
print(f"Min EPC (opportunity cost) = {min_EPC:,.2f}")
```

```

Optimal plan = (2, 2, 2)
Min EPC (opportunity cost) = 35,617.10
```

[10]:
```python
# Compute EUAC over study period under cash flow approach
EUAC_cf = -npf.pmt(marr, N, min_EPC-MV0_d, 0)
print(f"Optimal EUAC (cash flow) = {EUAC_cf:,.2f}")
```

```
Optimal EUAC (cash flow) = 7,029.91
```

```
[ ]:
```

10.5.2 Dynamic Programming Approach

Source: 8.5.2_replacement_analysis_finite_horizon_Dynamic_Programming.ipynb

```
[1]: # 8.5.2_replacement_analysis_finite_horizon_Dynamic_Programming.ipynb
""" 8.5.2 Optimal Replacement under Finite Planning Horizon
    Dynamic Programming Approach """
import pandas as pd
import numpy_financial as npf
from EngFinancialPy import Asset, pprint_list
```

```
[2]: # Global constants
marr = 0.1
study_period = 6
```

```
[3]: # Global data structures for optimal solution tracing
data = pd.DataFrame() # Assets data
decision = {}  # Decisions made at all nodes visited
opt_path = {}  # Optimnal decision path traced
```

```
[4]: def main():
    print("\nOptimal Replacement, Finite Study Period using Dynamic␣
 ↪Programming")
    print(f"Study period = {study_period} years")
    print(f"marr = {marr*100:.1f} %")

    Assets_Data()
    print(f"\nAvailable assets: {list(data.index)}")

    # initial states for Dynamic Programming
    year_now = 0
    current_asset = data.index[0]
    used = 0

    # Compute optimal solution using dynamic programming
    EPC_opp = DP(year_now, current_asset, used)

    # Trace the optimal decsions for each year
    Trace_optimal_solution(year_now, current_asset, used)
    print("\nOptimal Decisions:")
    for s in opt_path:
        print(f"  {s} -> {opt_path[s]}")

    # Display years of usage of each asset (k1, k2, k3)
    print("\nOptimal replacement plan:",
          [data["usage"][a] for a in data.index])
```

```
    for a in data.index:
        print(f"  Use {a} for {data['usage'][a]} years")

    # Display Optimal EPC and EUAC
    print(f"\nOptimal EPC (opportunity) = {EPC_opp:10,.2f}")
    EPC_cf = EPC_opp - data['MV0']['D0']
    print(f"Optimal EPC (cash flows)  = {EPC_cf:10,.2f}")
    EUAC_cf = -npf.pmt(marr, study_period, EPC_cf, 0)
    print(f"Optimal EUAC (cash flows) = {EUAC_cf:10,.2f}")
```

[5]:
```
# Set up data
def Assets_Data():
    global data

    # Defender data
    MV0_d = 5000
    MV_d = [4000, 3000, 2000, 1000 ]
    E_d = [5500, 6600, 7800, 8800 ]
    defender = Asset(MV0_d, MV_d, E_d, marr, name="Defender")
    TC_d = defender.TC()
    life_d = defender.useful_life()
    print(f"Defender remaining life = {life_d} years")
    pprint("Defender TC", TC_d)

    # Challenger data
    MV0_c = 20000
    MV_c = [15000, 11250, 8500, 6500,  4750 ]
    E_c = [ 2000,  3000, 4620, 8000, 12000 ]
    challenger = Asset(MV0_c, MV_c, E_c, marr, age=0, name="Challenger")
    TC_c = challenger.TC()
    life_c = challenger.useful_life()
    print(f"Challenger useful life = {life_c} years")
    pprint("Challenger TC", TC_c)
    econ_life_c, euac_star = challenger.econ_life_euac()
    print(f"Challenger econ life = {econ_life_c} yrs at EUAC*={euac_star:,.
 ↪2f}")

    # Put all available assets into a dictionary
    assets_data = { 'asset' : ["D0",   "C1",    "C2" ],
                    'TC'    : [ TC_d,   TC_c,   TC_c ],
                    'MV0'   : [ MV0_d,  MV0_c,  MV0_c],
                    'life'  : [ life_d, life_c, life_c],
                    'usage' : [ 0,  0,  0] }  # starting (k1, k2, k3)

    # Convert assets data to dataframe
    data = pd.DataFrame(assets_data)
    data = data.set_index('asset')
```

[6]:
```
# next available asset for replacement if any
def next_asset(mc):
    """ Determine next available asset for replacement if any """
```

```
    assets = list(data.index)
    head = assets.pop(0)
    while assets != []:
        if head == mc :
            return assets[0]
        else:
            head = assets.pop(0)
    return 'Nil'
```

[7]:
```
## Dynamic Programming
def DP(t, mc, used):
    """ Perform Dynamic Programming """
    global decision
    # Study period reached
    if t == study_period:
        decision[(t,mc,used)] = "End"
        return 0
    nex = next_asset(mc)
    if used == data['life'][mc] and nex == "Nil":
        # Current asset reached max life and no replacement is available
        decision[(t,mc,used)] = "Infeasible"
        return 10**10  # Big-M
    if used == data['life'][mc] and nex != "Nil":
        # Current asset reached max life and a replacement is available
        # Replace only
        replace_cost = data['TC'][nex][0]/(1+marr)**(t+1) + DP(t+1,nex,1)
        decision[(t,mc,used)] = "Replace"
        return replace_cost
    if used < data['life'][mc] and nex == "Nil":
        # Current asset has not reached max life and no replacement
        # is available. Therefore, Keep only
        keep_cost = data['TC'][mc][used]/(1+marr)**(t+1) + DP(t+1,mc,used+1)
        decision[(t,mc,used)] = "Keep"
        return keep_cost
    if used < data['life'][mc] and nex != "Nil":
        # Current asset has not reached max life and a replacement is␣
↪available
        # Choice of either Keep or Replace
        replace_cost = data['TC'][nex][0]/(1+marr)**(t+1) + DP(t+1,nex,1)
        keep_cost = data['TC'][mc][used]/(1+marr)**(t+1) + DP(t+1,mc,used+1)
        if keep_cost < replace_cost:
            decision[(t,mc,used)] = "Keep"
            return keep_cost
        else:
            decision[(t,mc,used)] = "Replace"
            return replace_cost
```

[8]:
```
# Trace the optimal replacement decisions
def Trace_optimal_solution(t, mc, used):
    """ Trace Optimal Replacement Decisions """
    global data
```

```python
    global opt_path
    state = (t, mc, used)
    action = decision[state]
    opt_path[state] = action
    # print(state, action)
    if action == "End": return
    if action == "Infeasible": return
    if action == "Keep":
        data.loc[mc, 'usage'] += 1
        Trace_optimal_solution(t+1, mc, used+1)
    if action == "Replace":
        data.loc[next_asset(mc), 'usage'] += 1
        Trace_optimal_solution(t+1, next_asset(mc), 1)
```

[9]:
```python
def pprint(label, list_of_numbers):
    """ Pretty format print a List of numbers """
    print(f"{label} =", "".join(f'{x:,.2f}   ' for x in list_of_numbers))
```

[10]:
```python
main()
```

```
Optimal Replacement, Finite Study Period using Dynamic Programming
Study period = 6 years
marr = 10.0 %
Defender remaining life = 4 years
Defender TC = 7,000.00   8,000.00   9,100.00   10,000.00
Challenger useful life = 5 years
Challenger TC = 9,000.00   8,250.00   8,495.00   10,850.00   14,400.00
Challenger econ life = 3 yrs at EUAC*=8,598.19

Available assets: ['D0', 'C1', 'C2']

Optimal Decisions:
  (0, 'D0', 0) -> Keep
  (1, 'D0', 1) -> Keep
  (2, 'D0', 2) -> Replace
  (3, 'C1', 1) -> Keep
  (4, 'C1', 2) -> Replace
  (5, 'C2', 1) -> Keep
  (6, 'C2', 2) -> End

Optimal replacement plan: [2, 2, 2]
  Use D0 for 2 years
  Use C1 for 2 years
  Use C2 for 2 years

Optimal EPC (opportunity) =  35,617.10
Optimal EPC (cash flows)  =  30,617.10
Optimal EUAC (cash flows) =   7,029.91
```

[ ]:

11 Learning Curve Model

11.1 Class LearningCurve

[1]:
```
from EngFinancialPy import LearningCurve
```

[2]:
```
print(LearningCurve.__doc__)
```

```
Learning Curve Model
   LearningCurve(K, s)
   Parameters:
     K = time/resource for first unit.
     s = learning curve parameter (0 < s < 1)
   Attributes:
     K = time/resource for first unit.
     s = learning curve parameter
   Methods:
     Unit(u): The time/resource required for unit u
     Cumulative(u): The cumulative time/resources for first u units
     Average(u): The average time/resource per unit for first u units
```

11.2 Example

Source: 9.5_learning_curve_model.ipynb

[1]:
```
# 9.5_learning_curve_model.ipynb
""" 9.5 Learning Curve Model """
from EngFinancialPy import LearningCurve
```

[2]:
```
# The time required to assemble the first car is 100 hours.
# The learning rate is 80%.
assemble_cars = LearningCurve(100, 0.8)
```

[3]:
```
# What is the time required to assemble the 10 car?
print("Time to assemble the 10th car = "
      f"{assemble_cars.Unit(10)}")
```

```
Time to assemble the 10th car = 47.65098748902245
```

[4]:
```
# What is the total time required to assemble the first 10 cars?
print("Time to assemble the first 10th car ="
      f"{assemble_cars.Cumulative(10)}")
```

```
Time to assemble the first 10th car =631.5373017615901
```

[5]:
```
# What is the average time per car for the first 10 car?
print("Average time per car for the first 10 cars = "
      f"{assemble_cars.Average(10)}")
```

```
Average time per car for the first 10 cars = 63.15373017615901 6
```

[6]:
```python
# What is the average time per car for the first 100 car?
print("Average time per car for the first 100 cars = "
      f"{assemble_cars.Average(100)}")
```

```
Average time per car for the first 100 cars = 32.65081083318006
```

[ ]:
```python
```