FIN514 Project 2 Choice 1

Auto Callable Yield Notes Linked to the S&P 500® Index

Spring 2022

Ya-Yen, Li(yayenli2)
Wei-Ting, Chao(wtchao3)
Yu-Shiuan, Chang(yschang4)

1. Executive Summary

We estimate the notes with the explicit finite difference method and Crank-Nicolson (CN) method with LU decomposition. Then we perform volatility and non-linearity analysis for both methods. For the continuous risk-free rate, we selected 0.007555; for dividends, we selected 1.455%; for implied volatility, we selected 21.59% (ATM at maturity). We select the volatilities on the final review date, ranging from 21.59% to 30.69%, and the values range from $960.80 to $991.25 per note. If the volatility equals 23.50%, the value will be approximately $983, which is the estimated value on the note.

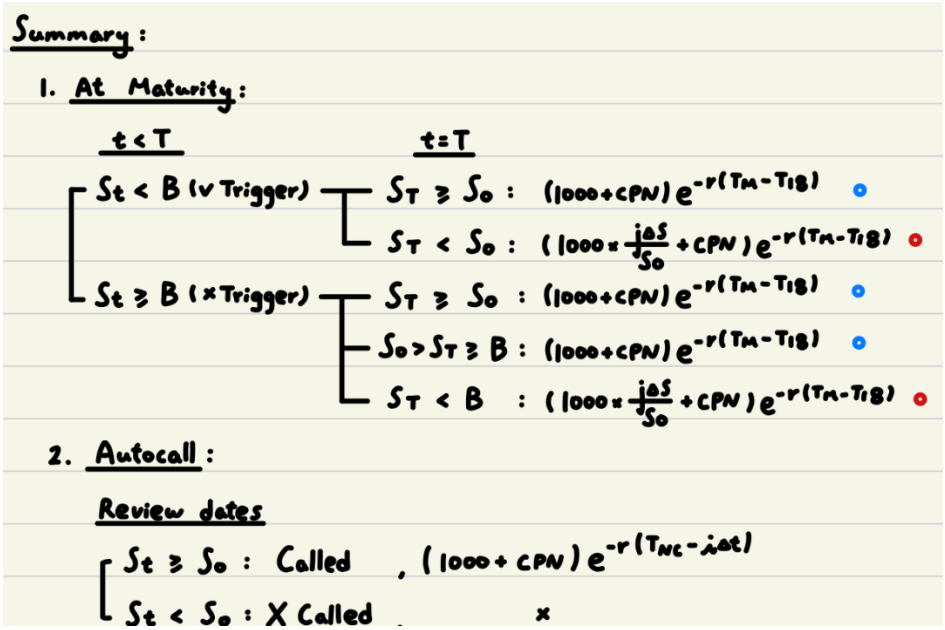
2. Key Feature

- ***Interest (coupon) payment:***
  If the product is not autocalled, the investor will receive a fixed payment of $5.375 per $1000 note for each month.

- ***Autocall:***
  If the closing level of the Index on any Review Date (other than the final Review Date) is greater or equal to the initial value (4577.11), the note will be automatically called to a cash payment and no further interest payment will be made.

- ***Trigger Event:***
  A trigger event will occur if the index of the S&P 500 is less than 70% of the initial value (Trigger = 0.7*S0= 3,203.977).

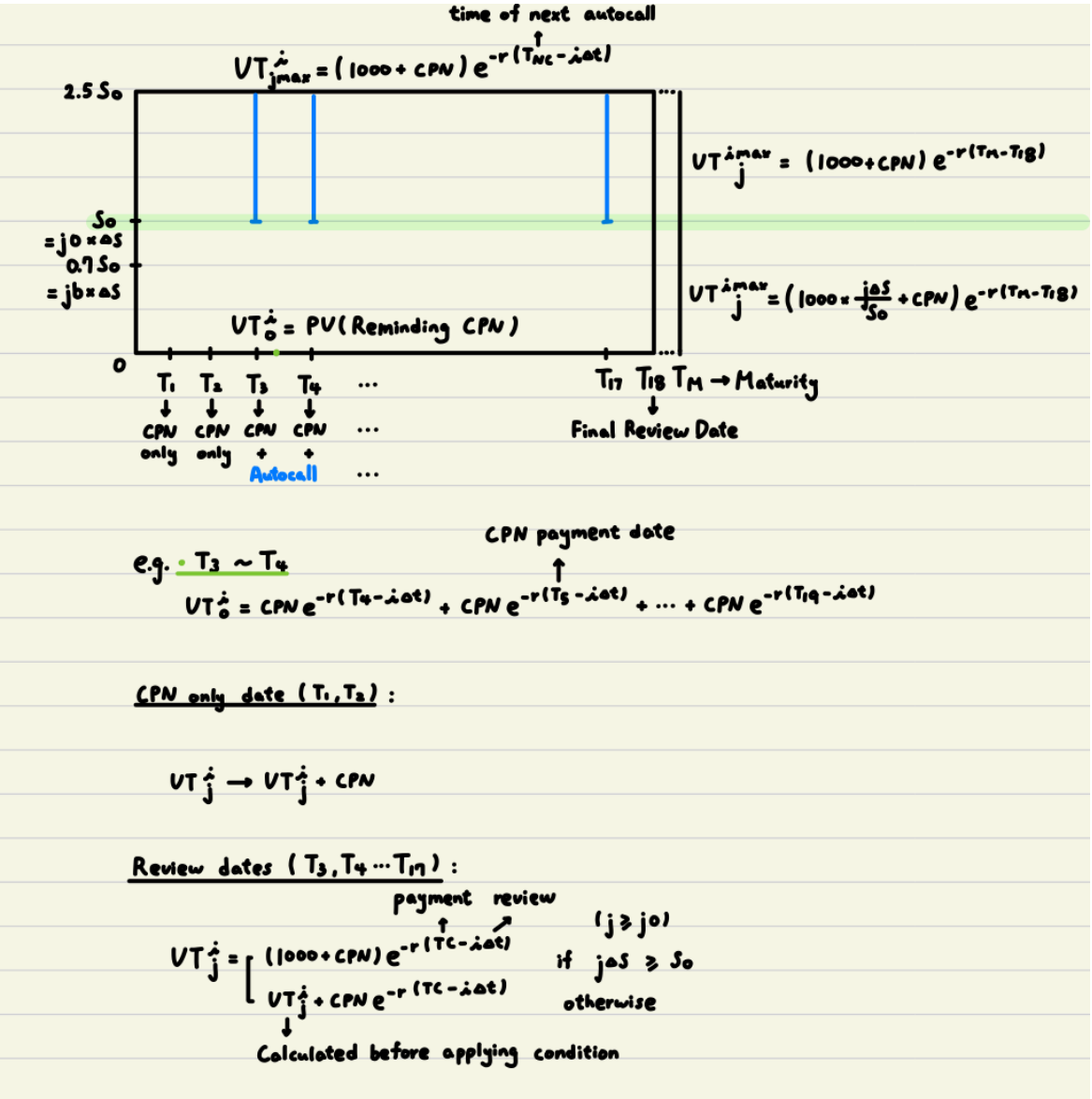
➢ ***Summarize the features:***



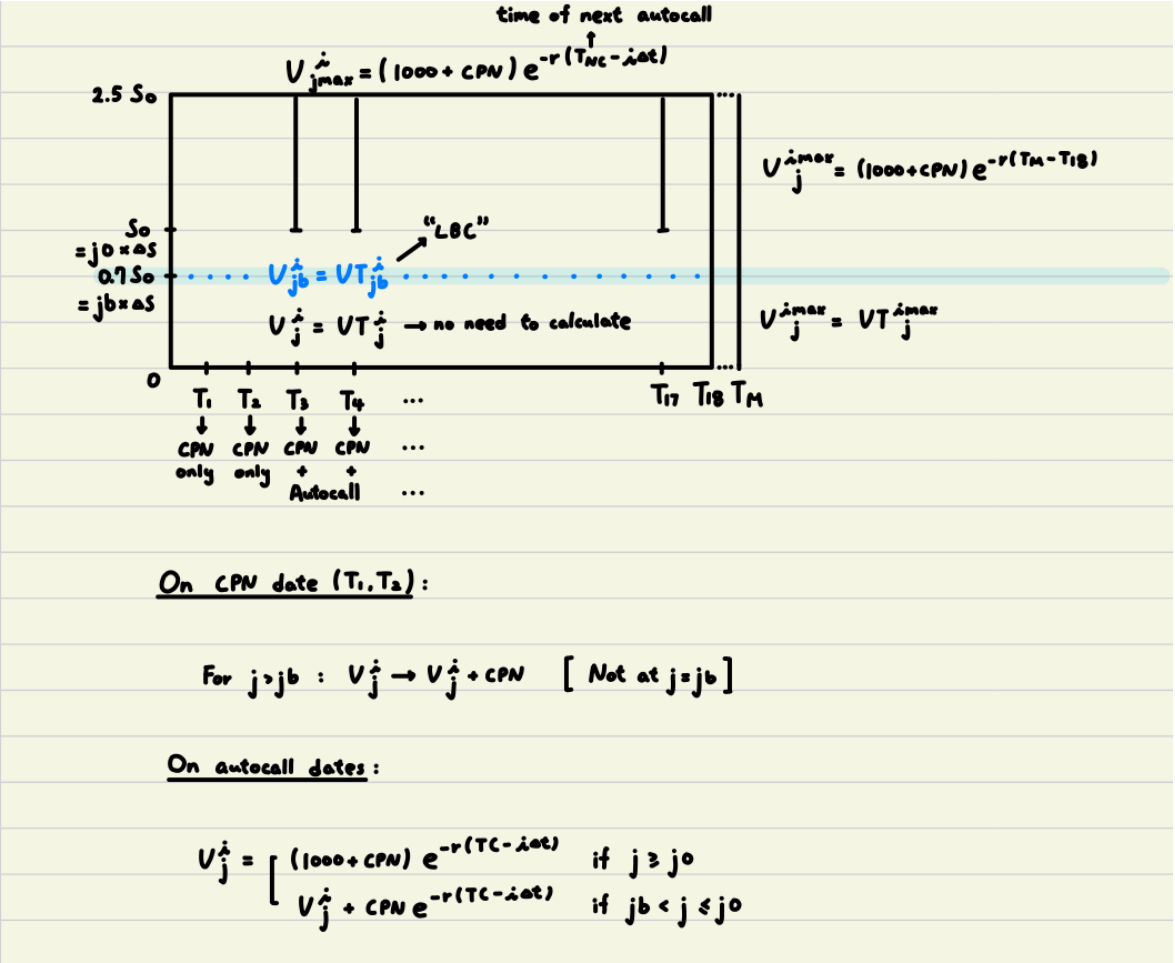
➢ ***Visualize the features:***

We build two grids. Assuming the trigger event does occur, we construct the VT grid. On the other hand, if the trigger event does not occur, we construct the V grid. Whenever the trigger event occurs, we replace the value in the V grid with those in the VT grid.

1. VT grid (Trigger event DOES occur):

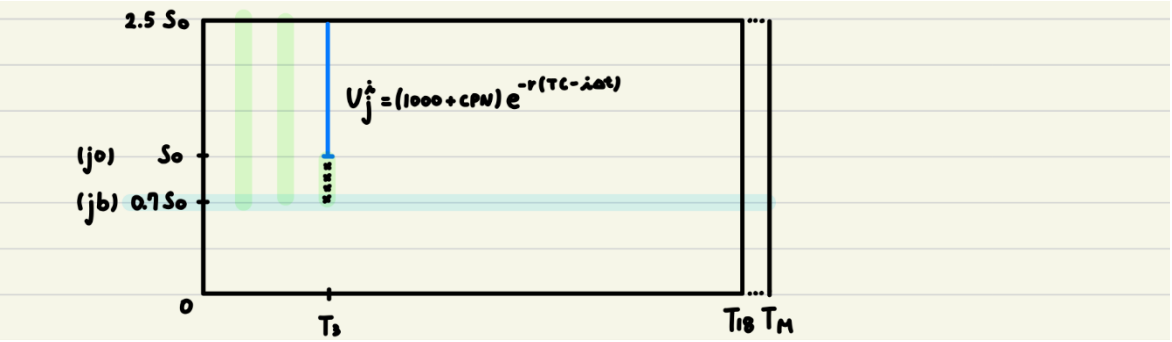


2. V grid (Trigger event does NOT occur):



3. Difference in CN method:

In VT / V grids, we only solve the green region on each auto call date.



- ***Timeline:***

	Pricing date	Settlement date																Final Review date	Maturity date
	1/18/2022	1/21/2022																7/18/2023	7/21/2023
T	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18 / TFR	T19 / TM
Review date			4/18/2022	5/18/2022	6/21/2022	7/18/2022	8/18/2022	9/19/2022	10/18/2022	11/18/2022	12/19/2022	1/18/2023	2/21/2023	3/20/2023	4/18/2023	5/18/2023	6/20/2023	7/18/2023	
Coupon date	2/24/2022	3/23/2022	4/21/2022	5/23/2022	6/24/2022	7/21/2022	8/23/2022	9/22/2022	10/21/2022	11/23/2022	12/22/2022	1/23/2023	2/24/2023	3/23/2023	4/21/2023	5/23/2023	6/23/2023		7/21/2023
Review to pricing date (Days)			90	120	154	181	212	244	273	304	335	365	399	426	455	485	518	546	
Coupon to pricing date (Days)	37	64	93	125	157	184	217	247	276	309	338	370	402	429	458	490	521		549

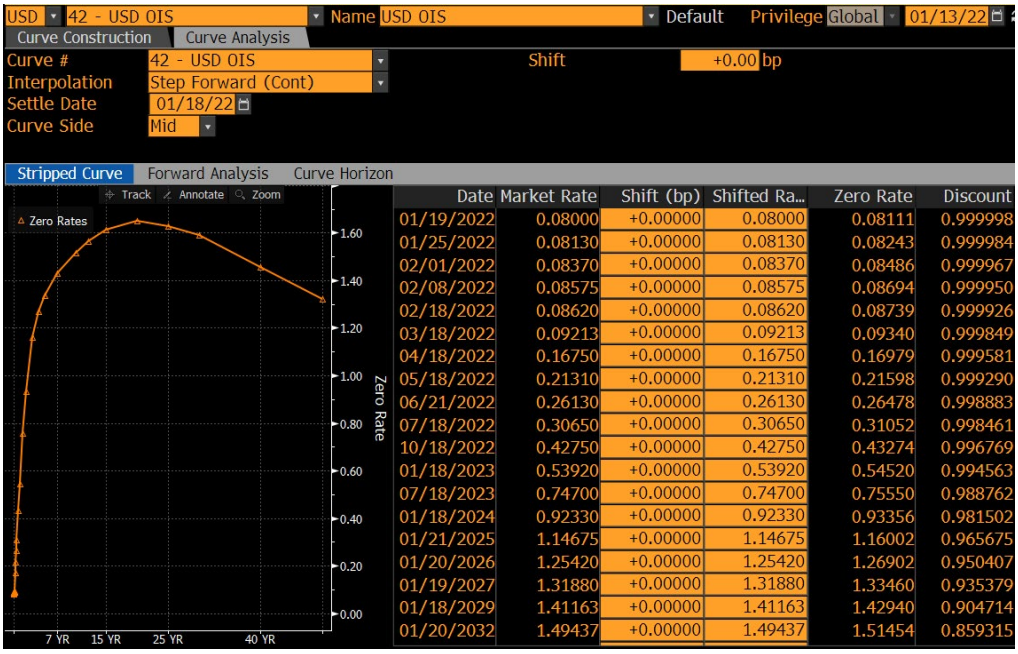
3. Valuation part 1 – Initiate

- ***Explicit Finite Difference vs. Crank-Nicolson:***

  The explicit finite difference method is unstable, and the convergence is only linear in t. On the other hand, Crank-Nicolson method will have a smaller time error since it is $(\Delta t)^2$ rather thanΔt . Most importantly there are no stability constraints, so we are free to choose any number of times, and S steps.

- ***Continuously compounded risk-free rate:***

  Quoted on 01/13/2022 (two days before trade date: 1/18/2022 and 01/17/2022 is Martin Luther King Memorial Day)



Date	Market Rate	Shift (bp)	Shifted Ra...	Zero Rate	Discount
01/19/2022	0.08000	+0.00000	0.08000	0.08111	0.999998
01/25/2022	0.08130	+0.00000	0.08130	0.08243	0.999984
02/01/2022	0.08370	+0.00000	0.08370	0.08486	0.999967
02/08/2022	0.08575	+0.00000	0.08575	0.08694	0.999950
02/18/2022	0.08620	+0.00000	0.08620	0.08739	0.999926
03/18/2022	0.09213	+0.00000	0.09213	0.09340	0.999849
04/18/2022	0.16750	+0.00000	0.16750	0.16979	0.999581
05/18/2022	0.21310	+0.00000	0.21310	0.21598	0.999290
06/21/2022	0.26130	+0.00000	0.26130	0.26478	0.998883
07/18/2022	0.30650	+0.00000	0.30650	0.31052	0.998461
10/18/2022	0.42750	+0.00000	0.42750	0.43274	0.996769
01/18/2023	0.53920	+0.00000	0.53920	0.54520	0.994563
07/18/2023	0.74700	+0.00000	0.74700	0.75550	0.988762
01/18/2024	0.92330	+0.00000	0.92330	0.93356	0.981502
01/21/2025	1.14675	+0.00000	1.14675	1.16002	0.965675
01/20/2026	1.25420	+0.00000	1.25420	1.26902	0.950407
01/19/2027	1.31880	+0.00000	1.31880	1.33460	0.935379
01/18/2029	1.41163	+0.00000	1.41163	1.42940	0.904714
01/20/2032	1.49437	+0.00000	1.49437	1.51454	0.859315

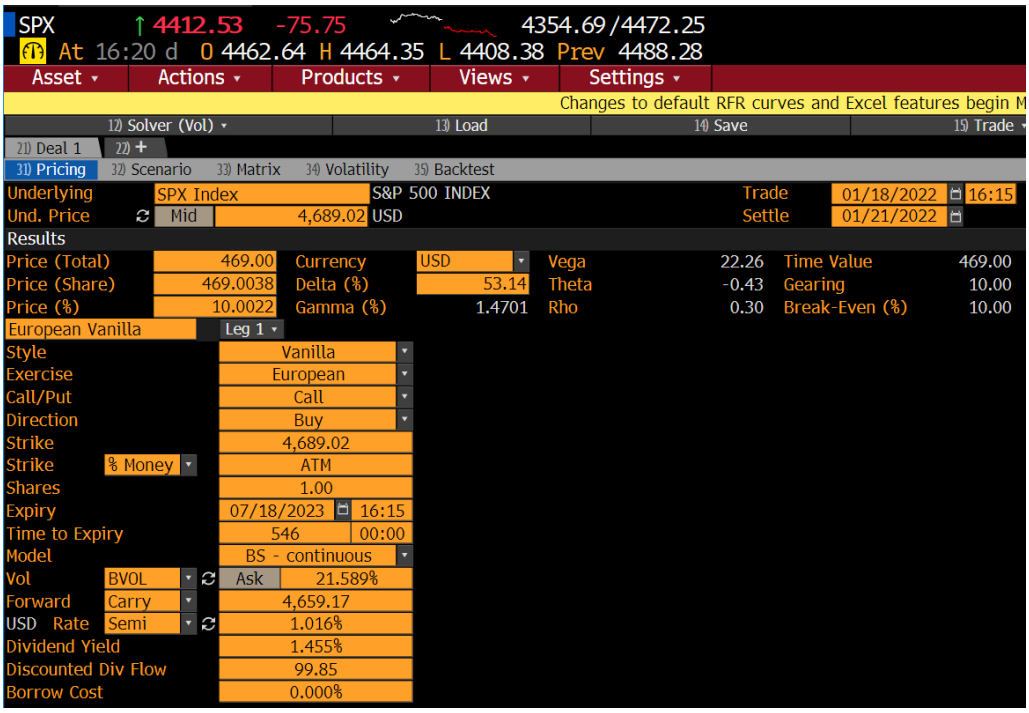
The continuously compounded risk-free rate (0.007555) is from the overnight index swap (OIS). Since the days are the same (546 days), we only need to transform discounted factor (DF) to continuously compounded (CC) where $CC = \frac{-\ln(DF)}{\frac{Days}{365}}$

Quote on 2022/01/13			Final Review Dates	Days	DF	CC
Prcing date	1/18/2022		7/18/2023	546	0.988762	0.007555
Maturity Date	**Days**	**Discount**				
01/19/2022	1	0.999998				
01/25/2022	7	0.999984				
02/01/2022	14	0.999967				
02/08/2022	21	0.99995				
02/18/2022	31	0.999926				
03/18/2022	59	0.999849				
04/18/2022	90	0.999581				
05/18/2022	120	0.99929				
06/21/2022	154	0.998883				
07/18/2022	181	0.998461				
10/18/2022	273	0.996769				
01/18/2023	365	0.994563				
07/18/2023	546	0.988762				
01/18/2024	730	0.981502				
01/21/2025	1099	0.965675				
01/20/2026	1463	0.950407				
01/19/2027	1827	0.935379				
01/18/2029	2557	0.904714				
01/20/2032	3654	0.859315				
01/18/2034	4383	0.828809				
01/20/2037	5481	0.785078				
01/21/2042	7308	0.718694				
01/18/2047	9131	0.665672				
01/18/2052	10957	0.620502				
01/18/2062	14610	0.558574				
01/19/2072	18263	0.516687				

- ***Continuously compounded dividends yield:***

  We use the continuously compounded dividend (1.455%) for S&P 500 since this index consists of many components and we consider this index to be paying out dividends every day.

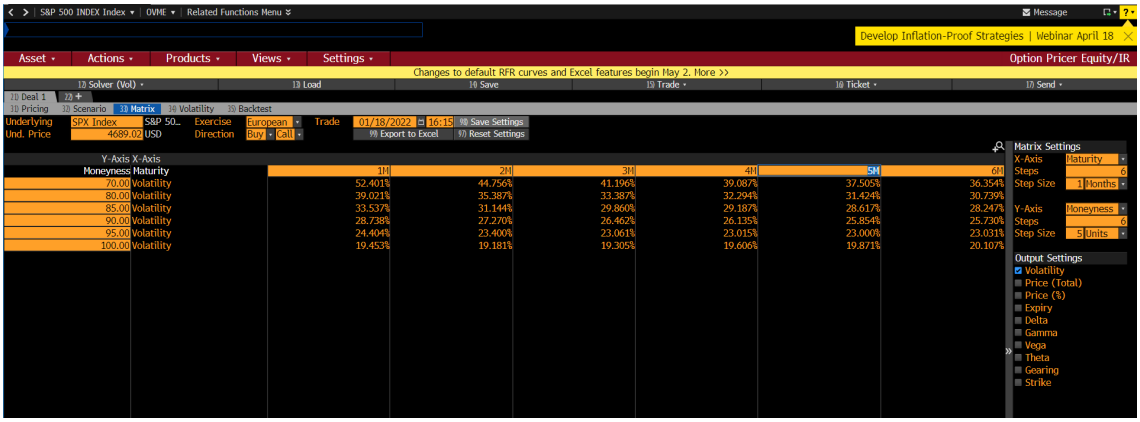
  (Trade date: 01/18/2022, Settlement date: 01/21/2022)

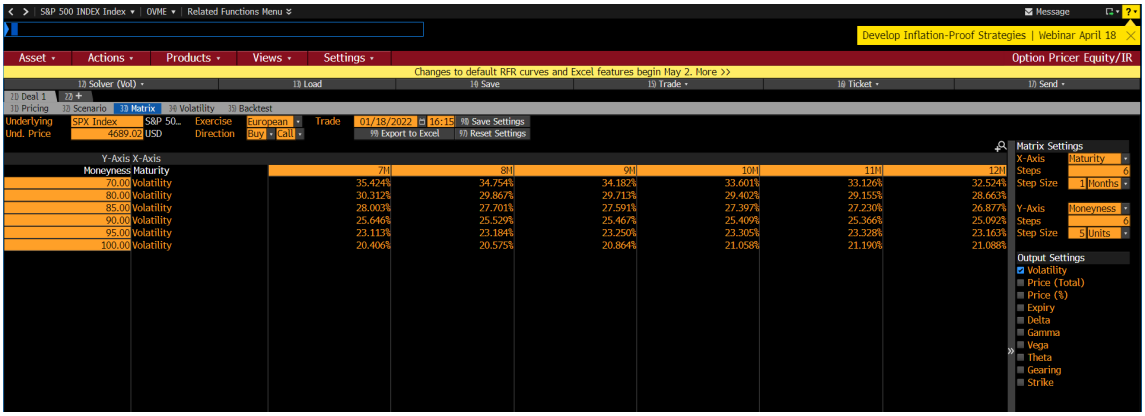


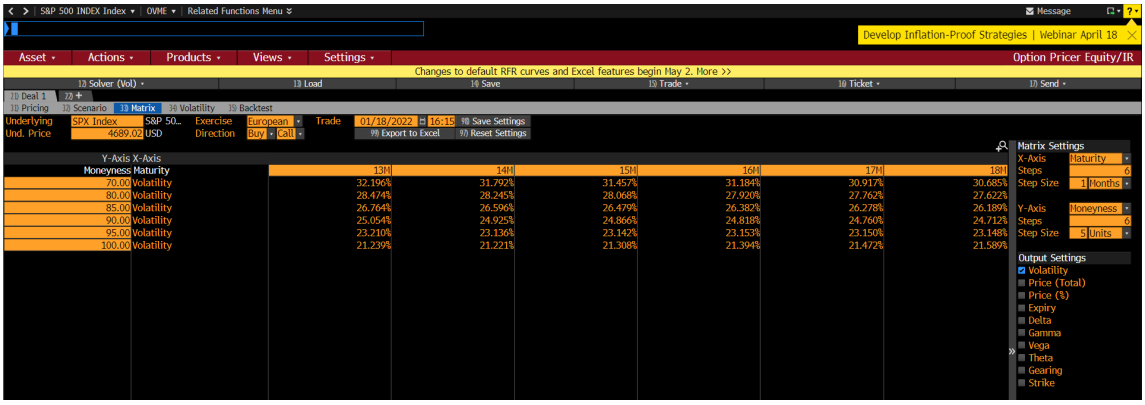
- ***Volatility***:

We get our volatility data from European options during the contract, then we calculate the implied volatility for each date that is relevant to our contract, which is the seventeen review dates plus one final review date. Moreover, we pick our moneyness from at the money to 0.7*S0 which is the trigger.

(Trade date: 01/18/2022)







In the end, we perform a volatility-sensitive analysis to test the probable range of product value. We choose the volatilities on the final review date, ranging from 21.59% to 30.69%.

Pricing date	1/18/2022															
Review Dates	4/18/2022	5/18/2022	6/18/2022	7/18/2022	8/18/2022	9/18/2022	10/18/2022	11/18/2022	12/18/2022	1/18/2023	2/18/2023	3/18/2023	4/18/2023	5/18/2023	6/18/2023	7/18/2023
Months	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Moneyness	Volatility															
70	41.20%	39.09%	36.35%	35.42%	35.42%	34.75%	34.18%	33.60%	33.13%	32.52%	32.20%	31.79%	31.46%	31.18%	30.92%	30.69%
80	33.39%	32.29%	31.42%	30.31%	30.31%	29.87%	29.71%	29.40%	29.16%	28.66%	28.47%	28.25%	28.07%	27.92%	27.76%	27.62%
85	29.86%	29.19%	28.62%	28.00%	28.00%	27.70%	27.59%	27.40%	27.23%	26.88%	26.76%	26.60%	26.48%	26.38%	26.28%	26.19%
90	26.46%	26.14%	25.85%	25.65%	25.65%	25.53%	25.47%	25.41%	25.37%	25.09%	25.05%	24.93%	24.87%	24.82%	24.76%	24.71%
95	23.06%	23.02%	23.00%	23.11%	23.11%	23.18%	23.25%	23.31%	23.33%	23.16%	23.21%	23.14%	23.14%	23.15%	23.15%	23.15%
100	19.31%	19.61%	19.87%	20.41%	20.41%	20.58%	20.86%	21.06%	21.19%	21.09%	21.24%	21.22%	21.31%	21.39%	21.47%	21.59%

4. Valuation part 2 – Explicit Finite Difference method

- ***Code – VT grid:***

At maturity:

If the final stock price is higher than or equal to the initial price, the notes will be auto called, so the value equals $1000 plus coupon then discounted from maturity date; otherwise, since we assume the trigger event **does** occur, the value is $1000 times index return plus the coupon then discounted from maturity date. And we also calculate probabilities A, B, C.

```
### At maturity
# Calculate option value at maturity VT[imax,j]
# Also calculate the probabilities A, B, C
i = imax

for j in range(0, jmax+1):

    if j >= j0:
        VT[i, j] = (1000 + CPN) * np.exp(TM - TFR)  # Autocall
    else:
        VT[i, j] = (1000 * (j*dS/S0) + CPN) * np.exp(TM - TFR)

    A[j] = (0.5 * (sigma ** 2) * (j ** 2) + 0.5 * (r - q) * j) * dt
    B[j] =  1 - r * dt - sigma ** 2 * j ** 2 * dt
    C[j] = (0.5 * (sigma ** 2) * (j ** 2) - 0.5 * (r - q) * j) * dt
```

Backward in time:

We split the process into three parts. Firstly, the lower boundary condition is the remaining coupons. Secondly, in the middle of the grid, the value equals VT[i, j] plus coupon on coupon-only dates (T1, T2). On review dates (T3~T17), if the stock value is higher than or equal to the initial stock price, the value equals $1000 plus coupon then discounted from coupon payment date; otherwise, it equals VT[i, j] plus coupon discounted from coupon payment date. Thirdly, the upper boundary condition equals $1000 plus coupon then discounted from next autocall date.

```
### Backward in time
counter1 = 0
counter2 = 0

for i in range(imax-1, -1, -1):

    # Lower boundary condition for VT
    # VT[i,0]=PV(remaining coupons)
    VT[i, 0] = 0

    # Check if i meet the review dates (T18 not included)
    if (i+1) in itotal:
        counter1 += 1
    if counter1 > 0:
        PV_cpn = []             # List to save the coupons
        for k in range(1, counter1+1):
            PV_cpn.append(CPN * np.exp(-r *(itotal[-k] * dt - i * dt)))  # Append each coupon in the list
        VT[i, 0] = sum(PV_cpn)  # Sum up the coupons
```

```
# Regular finite difference formula
for j in range(1, jmax, 1):

    VT[i,j] = A[j]*VT[i+1,j+1]+B[j]*VT[i+1,j]+C[j]*VT[i+1,j-1]

    # Payment dates for coupons T1,T2 (coupon only date)
    if i in icpn:
        VT[i,j] += CPN

    # Payment dates for coupons T3~T17
    if i in ireview:
        if j * dS >= S0:
            VT[i,j] = (1000 + CPN) * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
        else:
            VT[i,j] += CPN * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
            # ipayment[ireview.index(i)]: time of coupon date

# Upper boundary condition for VT
if (i+1) in ireview:  # If next time step is autocall date???
    counter2 += 1
VT[i,jmax] = (1000 + CPN) * np.exp(-r * (ireview[-counter2] * dt - i * dt))
# ireview[-counter2]: time of next autocall
```

- ***Code – V grid:***

At maturity:

If the final stock price is higher than or equal to the barrier, the value equals $1000 plus coupon then discounted from maturity date; otherwise, the value equals the value from the grid VT[imax,j].

```
### At maturity
# Calculate option value at maturity V[imax,j]
# Probabilities A, B, C are the same from V
i = imax

for j in range(0, jmax+1):
    if j >= jb:
        V[i, j] = (1000 + CPN) * np.exp(-r *(TM - TFR))
```

Backward in time:

We split the process into four parts. Firstly, the lower boundary condition is the barrier, so we get the value from grid VT[i,jb]. Secondly, in the middle of the grid, if the stock price is higher than the trigger, the value equals V[i, j] plus coupon on coupon-only dates (T1, T2). On review dates (T3~T17), if the stock value is higher than or equal to the initial stock price, the value equals $1000 plus coupon then discounted from the coupon payment date; if it is lower than the initial stock price but higher than the trigger, it equals V[i, j] plus coupon

discounted from coupon payment date. Thirdly, the upper boundary condition equals $1000 plus coupon then discounted from next autocall date. Fourthly, when the stock price is lower than the trigger we replace all V[i, j] with VT[i, j] since the trigger event does occur.

```
### Backward in time
for i in range(imax-1, -1, -1):

    # Lower boundary condition for V
    # Get value from VT
    V[i, jb] = VT[i, jb]

    # Regular finite difference formula
    for j in range(jb+1, jmax, 1):

        V[i,j] = A[j]*V[i+1,j+1]+B[j]*V[i+1,j]+C[j]*V[i+1,j-1]

        # Payment dates for coupons T1,T2 (coupon only date)
        if i in icpn:
            V[i,j] += CPN

        # Payment dates for coupons T3~T17
        if i in ireview:
            if j * dS >= S0:  # Autocall
                V[i,j] = (1000 + CPN) * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
            else:
                V[i,j] += CPN * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
                # ipayment[ireview.index(i)]: time of coupon date
```

```
    # Upper boundary condition for V
    if (i+1) in ireview:
        counter3 += 1
    V[i,jmax] = (1000 + CPN) * np.exp(-r * (ireview[-counter3] * dt - i * dt))
    # ireview[-counter3]: time of next autocall

# If value lower than LBC, then get value from VT
V[:, 0:jb] = VT[:, 0:jb]
```

- ***Result:***

  With all the parameters and the explicit finite difference model, we choose, we get our product value to be $991.27 per note.

```
1 expfd = EXPFD(S0, Trigger, TM, TFR, r, q, sigma, SU, imax, jmax, CPN, Review_dates, Payment_dates, CPN_payment_dates)
2 print(expfd)
```

```
991.27041730021
```

5. Valuation part 3 – Crank-Nicolson method with LU decomposition

- ***Code – At maturity:***

At maturity:

If the final stock price is higher than or equal to the initial price, the notes will be auto called, so the value equals $1000 plus coupon then discounted from maturity date; otherwise, since we assume the trigger event **does** occur, the value is $1000 times index return plus the coupon then discounted from maturity date.

```
### At maturity
# Calculate option value at maturity VT[imax,j]
i = imax

for j in range(0, jmax+1):

    if j >= j0:
        VT[i, j] = (1000 + CPN) *  np.exp(-r *(TM - TFR))
    else:
        VT[i, j] = (1000 * (j*dS/S0) + CPN) *  np.exp(-r *(TM - TFR))
```

Backward in time:

The boundaries conditions are similar to the explicit finite method of the VT grid. However, while using the LU decomposition, we made some adjustments to avoid errors, especially on review dates. On review dates, the upper boundary becomes j0 and we add the coupon at last. In this way, we can avoid adding coupons twice.

```
### Backward in time
counter1 = 0   # Counter for remaining coupons
counter2 = 0   # Counter for next autocall day

for i in range(imax-1, -1, -1):

    ### On review days
    if i in ireview:
        ### Lower boundary condition in matrix terms
        A[0] = 0
        B[0] = 1
        C[0] = 0

        # D[0]=PV(remaining coupons)
        PV_cpn = []        # List to save the dicounted coupons
        remain_cpn = len(itotal) - (ireview.index(i)+2)  # Quantities of remaining coupon
        for k in range(1, remain_cpn):
            PV_cpn.append(CPN * np.exp(-r *(itotal[-k] * dt - i * dt)))  # Append dicounted coupons in the list
        D[0] = sum(PV_cpn) # Sum up the dicounted coupons
```

```
      # Regular D[j] values
      # Exclude D[0], D[j0]
      # Also calculate the probabilities A, B and C, and D for each j step
      for j in range(1, j0, 1):
          A[j] = 0.25*sigma**2*j**2 - 0.25*(r-q)*j
          B[j] = -1/dt-0.5*r-0.5*sigma**2*j**2
          C[j] = 0.25*sigma**2*j**2 + 0.25*(r-q)*j
          D1 = -VT[i+1,j-1]*(0.25*sigma**2*j**2-0.25*(r-q)*j)
          D2 = -VT[i+1,j]*(1/dt-0.5*r-0.5*sigma**2*j**2)
          D3 = -VT[i+1,j+1]*(0.25*sigma**2*j**2+0.25*(r-q)*j)
          D[j] = D1+D2+D3

      ### Upper boundary condition at j0 in matrix terms
      A[j0] = 0
      B[j0] = 1
      C[j0] = 0
      D[j0] = (1000 + CPN) * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
```

```
    ### Start LU decomposition
    alpha[0] = B[0]
    CN_S[0] = D[0]

    # Solve alpha, CN_S only from 1 to j0
    for j in range(1, j0+1, 1):
        alpha[j] = B[j]-(A[j]*C[j-1])/alpha[j-1]
        CN_S[j] = D[j]-(A[j]*CN_S[j-1])/alpha[j-1]
    VT[i,j0] = CN_S[j0]/alpha[j0]

    for j in range(j0-1, -1, -1):
        VT[i,j] = (CN_S[j]-C[j]*VT[i,j+1])/alpha[j]

    # Add the coupons at last
    for j in range(j0+1,jmax+1):
        VT[i,j] = (1000+CPN) * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))

    for j in range(1,j0):
        VT[i,j] += CPN * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
```

```
else:

    ### Lower boundary condition in matrix terms
    A[0] = 0
    B[0] = 1
    C[0] = 0
    D[0] = 0

    # Check if i meet the review dates (T18 not included)
    # D[0]=PV(remaining coupons)
    if (i+1) in itotal:
        counter1 += 1
    if counter1 > 0:
        PV_cpn = []         # List to save the dicounted coupons
        for k in range(1, counter1+1):
            PV_cpn.append(CPN * np.exp(-r *(itotal[-k] * dt - i * dt)))   # Append dicounted coupons in the list
        D[0] = sum(PV_cpn)  # Sum up the dicounted coupons

    # Regular D[j] values
    # Exclude D[0], D[jmax]
    # Also calculate the probabilities A, B and C, and D for each j step
    for j in range(1, jmax, 1):
        A[j] = 0.25*sigma**2*j**2 - 0.25*(r-q)*j
        B[j] = -1/dt-0.5*r-0.5*sigma**2*j**2
        C[j] = 0.25*sigma**2*j**2 + 0.25*(r-q)*j
        D1 = -VT[i+1,j-1]*(0.25*sigma**2*j**2-0.25*(r-q)*j)
        D2 = -VT[i+1,j]*(1/dt-0.5*r-0.5*sigma**2*j**2)
        D3 = -VT[i+1,j+1]*(0.25*sigma**2*j**2+0.25*(r-q)*j)
        D[j] = D1+D2+D3
```

```
    ### Upper boundary condition in matrix terms
    A[jmax] = 0
    B[jmax] = 1
    C[jmax] = 0
    if (i+1) in ireview:
        counter2 += 1
    D[jmax] = (1000 + CPN) * np.exp(-r * (ireview[-counter2] * dt - i * dt))
    # ireview[-counter2]: time of next autocall

    ### Start LU decomposition
    alpha[0] = B[0]
    CN_S[0] = D[0]


    # Solve alpha, CN_S other than autocall dates
    for j in range(1, jmax+1, 1):
        alpha[j] = B[j]-(A[j]*C[j-1])/alpha[j-1]
        CN_S[j] = D[j]-(A[j]*CN_S[j-1])/alpha[j-1]
    VT[i,jmax] = CN_S[jmax]/alpha[jmax]

    for j in range(jmax-1, -1, -1):
        VT[i,j] = (CN_S[j]-C[j]*VT[i,j+1])/alpha[j]

    # Add the coupons at last
    if i in icpn:
        for j in range(1, jmax):
            VT[i,j] += CPN * np.exp(-r * (itotal[icpn.index(i)] * dt - i * dt))
            # Add coupon on only coupon payment days
```

- ***Code – V grid:***

At maturity:

If the final stock price is higher than or equal to the trigger, the value equals $1000 plus coupon then discounted from maturity date; otherwise, the value equals the value from the grid VT[imax,j].

```
### At maturity
# Calculate option value at maturity V[imax,j]
i = imax

for j in range(jb, jmax+1):
    V[i, j] = (1000 + CPN) *  np.exp(-r *(TM - TFR))
```

Backward in time:

The boundaries conditions are similar to the explicit finite method of the V grid. However, while using the LU decomposition, we made some adjustments to avoid errors, especially on review dates. On review dates, the upper boundary becomes j0, the lower boundary is jb, and we add the coupon at last. In this way, we can avoid adding coupons twice. Lastly, when the stock price is lower than the trigger we replace all V[i, j] with VT[i, j] since the trigger event does occur.

```
### Backward in time
counter3 = 0   # Counter for next autocall day

#Now go back in time
for i in range(imax-1, -1, -1):

    ### On review days
    if i in ireview:
        ### Lower boundary condition at jb in matrix terms
        A[jb] = 0
        B[jb] = 1
        C[jb] = 0
        D[jb] = VT[i, jb]

        # Regular D[j] values
        # Exclude D[jb], D[j0]
        # Also calculate the probabilities A, B and C, and D for each j step
        for j in range(jb+1, j0, 1):
            A[j] = 0.25*sigma**2*j**2 - 0.25*(r-q)*j
            B[j] = -1/dt-0.5*r-0.5*sigma**2*j**2
            C[j] = 0.25*sigma**2*j**2 + 0.25*(r-q)*j
            D1 = -V[i+1,j-1]*(0.25*sigma**2*j**2-0.25*(r-q)*j)
            D2 = -V[i+1,j]*(1/dt-0.5*r-0.5*sigma**2*j**2)
            D3 = -V[i+1,j+1]*(0.25*sigma**2*j**2+0.25*(r-q)*j)
            D[j] = D1+D2+D3
```

```
        ### Upper boundary condition at j0 in matrix terms
        A[j0] = 0
        B[j0] = 1
        C[j0] = 0
        D[j0] = (1000 + CPN) * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))

        ### Start LU decomposition
        alpha[jb] = B[jb]
        CN_S[jb] = D[jb]

        # Solve alpha, CN_S only from 1 to j0
        for j in range(1, j0+1, 1):
            alpha[j] = B[j]-(A[j]*C[j-1])/alpha[j-1]
            CN_S[j] = D[j]-(A[j]*CN_S[j-1])/alpha[j-1]
        V[i,j0] = CN_S[j0]/alpha[j0]

        for j in range(j0-1, -1, -1):
            V[i,j] = (CN_S[j]-C[j]*V[i,j+1])/alpha[j]

        # Add the coupons at last
        for j in range(j0+1,jmax+1):
            V[i,j] = (1000+CPN) * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))

        for j in range(1,j0):
            V[i,j] += CPN * np.exp(-r * (ipayment[ireview.index(i)] * dt - i * dt))
```

```
    else:

        ### Lower boundary condition at jb in matrix terms
        A[jb] = 0
        B[jb] = 1
        C[jb] = 0
        D[jb] = VT[i, jb]

        # Regular D[j] values
        # Exclude D[jb], D[jmax]
        # Also calculate the probabilities A, B and C, and D for each j step
        for j in range(jb+1, jmax, 1):
            A[j] = 0.25*sigma**2*j**2 - 0.25*(r-q)*j
            B[j] = -1/dt-0.5*r-0.5*sigma**2*j**2
            C[j] = 0.25*sigma**2*j**2 + 0.25*(r-q)*j
            D1 = -V[i+1,j-1]*(0.25*sigma**2*j**2-0.25*(r-q)*j)
            D2 = -V[i+1,j]*(1/dt-0.5*r-0.5*sigma**2*j**2)
            D3 = -V[i+1,j+1]*(0.25*sigma**2*j**2+0.25*(r-q)*j)
            D[j] = D1+D2+D3
```

```
        ### Upper boundary condition in matrix terms
        A[jmax] = 0
        B[jmax] = 1
        C[jmax] = 0
        if (i+1) in ireview:
            counter3 += 1
        D[jmax] = (1000 + CPN) * np.exp(-r * (ireview[-counter3] * dt - i * dt))
        # ireview[-counter3]: time of next autocall

        ### Start LU decomposition
        alpha[jb] = B[jb]
        CN_S[jb] = D[jb]

        # Solve alpha, CN_S other than autocall dates
        for j in range(1, jmax+1, 1):
            alpha[j] = B[j]-(A[j]*C[j-1])/alpha[j-1]
            CN_S[j] = D[j]-(A[j]*CN_S[j-1])/alpha[j-1]
        V[i,jmax] = CN_S[jmax]/alpha[jmax]

        for j in range(jmax-1, -1, -1):
            V[i,j] = (CN_S[j]-C[j]*V[i,j+1])/alpha[j]

        # Add the coupons at last
        if i in icpn:
            for j in range(0, jmax):
                V[i,j] += CPN * np.exp(-r * (itotal[icpn.index(i)] * dt - i * dt))
                # Add coupon on only coupon payment days

# Replace V with VT when j belows jb
V[:, 0:jb] = VT[:, 0:jb]
```

- ***Result:***

  With all the parameters and the Crank-Nicolson model we choose, we get our product value to be $991.28 per note.

```
1 cnfd = CNFD(S0, Trigger, TM, TFR, r, q, sigma, SU, imax, jmax, CPN, Review_dates, Payment_dates, CPN_payment_dates)
2 cnfd
```

```
991.2896259469915
```

6. Volatility-sensitive analysis – Explicit Finite Difference method

- ***Volatility on different dates:***

  To analyze our choice of volatility, we compare our results with the volatility range from 100% (at-the-money) to 70% moneyness. Since the volatilities in each month are similar, we select those on final review date (18 months from the pricing date 07/18/2022).

```
# Volatities from 70~100 moneyness 18 months from pricing notes
vol = [0.3069,  # moneynes 70
       0.2762,  # moneynes 85
       0.2619,  # moneynes 80
       0.2471,  # moneynes 90
       0.2315,  # moneynes 95
       0.2159]  # moneynes 100 (ATM)
```

Moreover, we adjust the time steps to ensure stability.

```
# List to save the value
value_result = []

# Value with different volatilities
for i in vol:

    # Make sure the grid is stable
    if imax < TFR * i**2 * jmax**2:
        imax_new = int(TFR * i**2 * jmax**2) + 1000
    else:
        imax_new = imax

    note_value = EXPFD(S0, Barrier, TM, TFR, r, q, i, SU, imax_new, jmax, CPN, Review_dates, Payment_dates, CPN_payment_
    output = {'Volatility': i, 'Value': note_value}
    value_result.append(output)
```

- ***Results:***

  The values range from $960.80 to $991.25 per note while the volatilities range from 0.3069 to 0.2159. According to the product description, the product's value is $983 which could be selecting volatility between 23% and 24%.

```
# Print value with different volatilities
value_result
```

```
[{'Volatility': 0.3069, 'Value': 960.8096215552146},
 {'Volatility': 0.2762, 'Value': 969.9014951609474},
 {'Volatility': 0.2619, 'Value': 974.4958014944215},
 {'Volatility': 0.2471, 'Value': 979.5418295529953},
 {'Volatility': 0.2315, 'Value': 985.2125213722752},
 {'Volatility': 0.2159, 'Value': 991.2571601457116}]
```

7. Volatility-sensitive analysis – Crank-Nicolson method

- ***Volatility on different dates:***

  To analyze our choice of volatility, we compare our results with the volatility range from 100% (at-the-money) to 70% moneyness. Since the volatilities in each month are similar, we select those on final review date (18 months from the pricing date 07/18/2022).

```
# Volatities from 70~100 moneyness 18 months from pricing notes
vol = [0.3069,  # moneynes 70
       0.2762,  # moneynes 85
       0.2619,  # moneynes 80
       0.2471,  # moneynes 90
       0.2315,  # moneynes 95
       0.2159]  # moneynes 100 (ATM)
```

```
# List to save the value
value_result = []

# Value with different volatilities
for sigma in vol:
    note_value = CNFD(S0, Trigger, TM, TFR, r, q, sigma, SU, imax, jmax, CPN, Review_dates, Payment_dates, CPN_payment_d
    output = {'Volatility': sigma, 'Value': note_value}
    value_result.append(output)
```

- ***Results:***

  The values range from $960.89 to $991.28 per note while the volatilities range from 0.3069 to 0.2159. According to the product description, the product's value is $983 which could be selecting volatility between 23% and 24%.

```
# Print value with different volatilities
value_result
```

```
[{'Volatility': 0.3069, 'Value': 960.8982756438841},
 {'Volatility': 0.2762, 'Value': 969.9673392132244},
 {'Volatility': 0.2619, 'Value': 974.5518187914183},
 {'Volatility': 0.2471, 'Value': 979.5882818732235},
 {'Volatility': 0.2315, 'Value': 985.2496047557171},
 {'Volatility': 0.2159, 'Value': 991.2856389103231}]
```

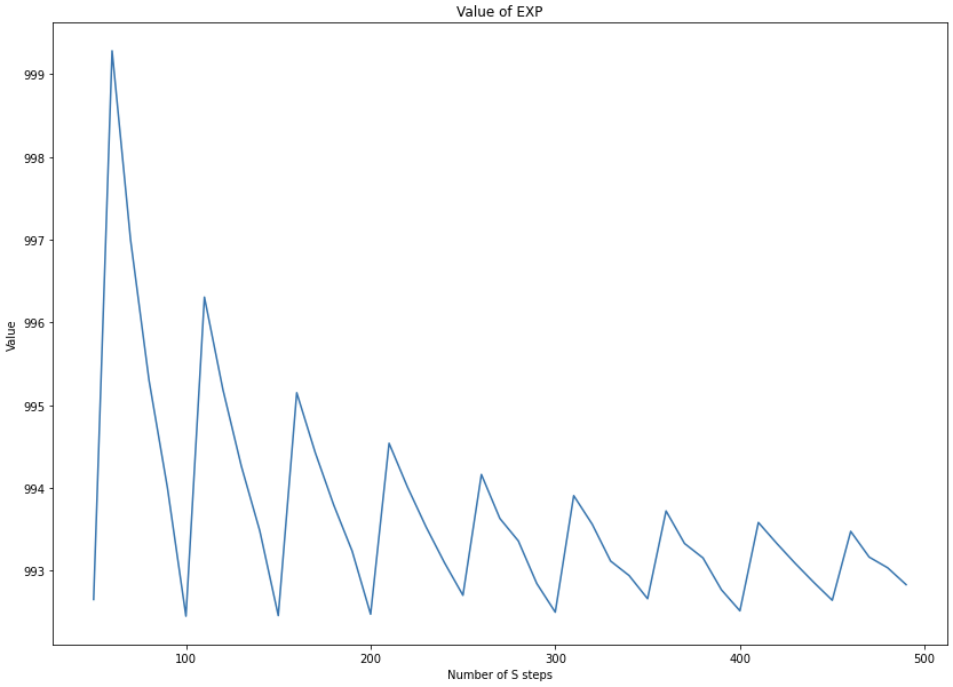
8. Non-linearity analysis – Explicit Finite Difference method

- ***Summary:***

  To check the non-linearity error, we perform two tests. In the first test, we vary our lambda by making our steps 10. In the second test, since the continuous barrier (Autocall) has the largest error, we can make the grid exactly on the notes by selecting lambda equals to 1 to reduce non-linearity error. The steps are 25 since the upper boundary is 2.5*S0. The range for each test is from 50 to 500. We check whether the value will slowly converge to a specific value.

- ***Varying Lambda:***

  In the non-linearity plot with different Lambda, the value will slowly converge to $992.5.



Tables:

```
1 df_varyLamda.head()
```

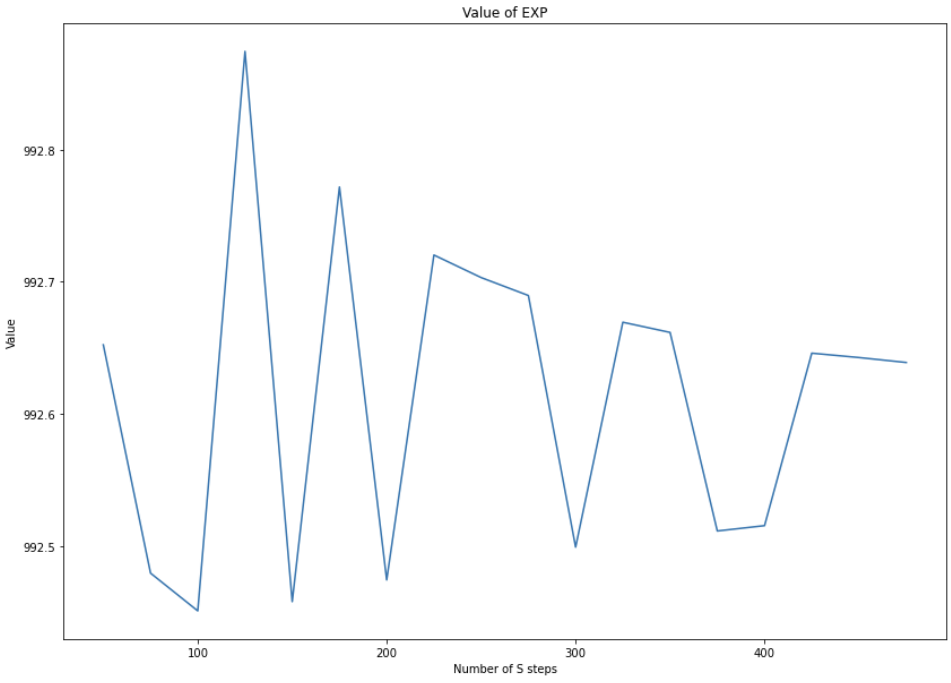
	S_steps	t_steps	EXP	Lambda
0	50	10000	992.652273	1.0
1	60	10000	999.280919	0.2
2	70	10000	996.998046	0.4
3	80	10000	995.300280	0.6
4	90	10000	993.998623	0.8

```
1 df_varyLamda.tail()
```

	S_steps	t_steps	EXP	Lambda
40	450	15118	992.642641	1.0
41	460	15753	993.475367	0.2
42	470	16401	993.164876	0.4
43	480	17063	993.035546	0.6
44	490	17740	992.831778	0.8

- ***Lambda equals 1:***

  Since the continuous barrier (Autocall) has the largest error, we can make the grid exactly on the notes by selecting Lambda equals to 1. The value will slowly converge to $992.6.



Tables:

```
df_Lamda1.head()
```

	S_steps	t_steps	EXP	Lambda
0	50	10000	992.652273	1.0
1	75	10000	992.479427	1.0
2	100	10000	992.450831	1.0
3	125	10000	992.874428	1.0
4	150	10000	992.457853	1.0

```
df_Lamda1.tail()
```

	S_steps	t_steps	EXP	Lambda
13	375	10000	992.511342	1.0
14	400	12155	992.515374	1.0
15	425	13593	992.645946	1.0
16	450	15118	992.642641	1.0
17	475	16730	992.638893	1.0

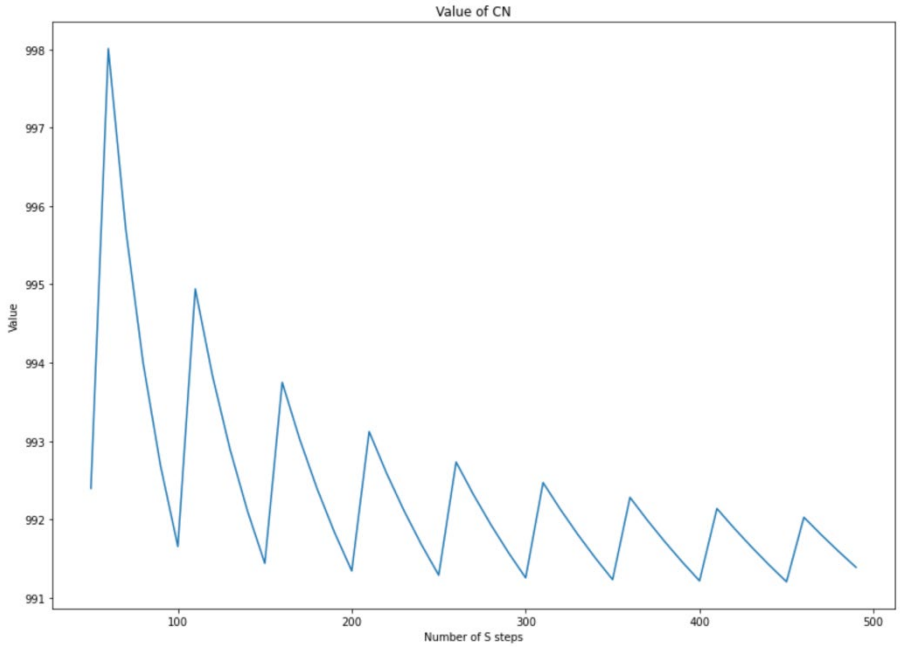
9. Non-linearity analysis – Crank-Nicolson method

- ***Summary:***

  To check the non-linearity error, we perform two tests. In the first test, we vary our lambda by making our steps 10. In the second test, since the continuous barrier (Autocall) has the largest error, we can make the grid exactly on the notes by selecting lambda equals to 1 to reduce non-linearity error. The steps are 25 since the upper boundary is 2.5*S0. The range for each test is from 50 to 500. We check whether the value will slowly converge to a specific value.

- ***Varying Lambda:***

  In the non-linearity plot with different Lambda, the value will slowly converge to $991.5.



Tables:

```
1 df_varyLamda.head()
```

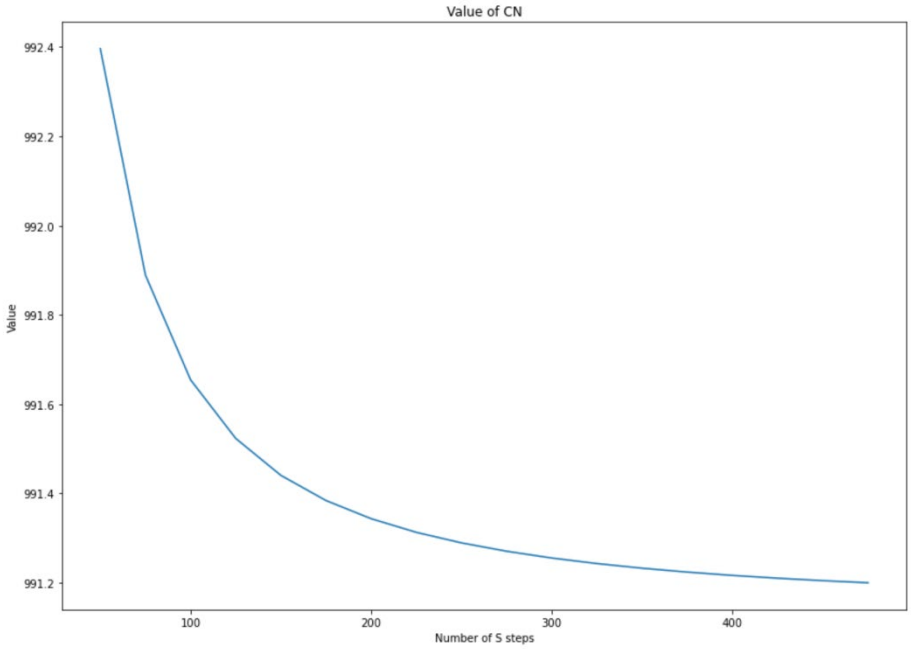
	S_steps	t_steps	CN	Barrier Lambda
0	50	10000	992.396227	1.0
1	60	10000	998.008424	0.2
2	70	10000	995.710350	0.4
3	80	10000	993.999865	0.6
4	90	10000	992.687585	0.8

```
1 df_varyLamda.tail()
```

	S_steps	t_steps	CN	Barrier Lambda
40	450	10000	991.204339	1.0
41	460	10000	992.027593	0.2
42	470	10000	991.804102	0.4
43	480	10000	991.591399	0.6
44	490	10000	991.388759	0.8

- ***Lambda equals 1:***

  Since the continuous barrier (Autocall) has the largest error, we can make the grid exactly on the notes by selecting Lambda equals to 1. The value will slowly converge to $991.2.



Tables:

```
1 df_Lamda1.head()
```

	S_steps	t_steps	CN	Barrier Lambda
0	50	10000	992.396227	1.0
1	75	10000	991.888823	1.0
2	100	10000	991.654115	1.0
3	125	10000	991.522855	1.0
4	150	10000	991.440150	1.0

```
1 df_Lamda1.tail()
```

	S_steps	t_steps	CN	Barrier Lambda
13	375	10000	991.223508	1.0
14	400	10000	991.216113	1.0
15	425	10000	991.209786	1.0
16	450	10000	991.204339	1.0
17	475	10000	991.199623	1.0

10. Conclusion

For this project, we applied two methods, which are the explicit finite difference method and the Crank-Nicolson method, to price this structured note. The values we estimate under these two methods are nearly identical; however, it is about $10 different from the estimated value from the note. We believe the reason that causes this difference is the volatility we choose. Since the interpretation of volatility varies from people; nevertheless, we have a range of volatilities that include the estimated value of $983. As for the two pricing methods, the explicit finite difference method required us to check the stability with different volatility; on the other hand, Crank-Nicolson method is much more stable and the non-linearity plot

is smoother than explicit finite difference method.

Reference

Auto Callable Yield Notes Linked to the S&P 500® Index, due July 23, 2023 (JPMorgan Chase Financial Company LLC)
https://www.sec.gov/Archives/edgar/data/0001665650/000182912622001257/jpm_424b2.htm