14 单因子利率模型三叉树

被使用申明*

2021 年 2 月 1 日

目录

1 简介 **1**
1.1 Hull-White 单因子模型 1
1.2 三叉树利率模型 1
1.3 零息债券期权 2

2 生成树形和计算债券期权价格 **2**
2.1 生成利率三叉树 2
2.2 计算欧式零息债券期权价格 3

3 步骤 C 代码实现 **3**

4 计算示例 **9**

5 参考资料 **10**

1 简介

1.1 Hull-White 单因子模型

Hull-White 单因子模型是一种描述瞬时无风险利率变化过程的模型。该模型计算出的初始利率期限结构可以与市场上观察到的利率期限结果相吻合。而且利率在变化过程中具有均值回归特性，即当瞬时无风险利率偏离初始利率期限结构时，它会以一定的速度回归到初始利率期限结构。模型可以被表示为

$$dr(t) = [\theta(t) - ar(t)]dt + \sigma dz \ . \tag{1}$$

其中 $r(t)$ 为在 t 时刻的瞬时无风险利率，a 和 σ 为常数，$\theta(t)$ 为由初始利率期限结构所确定的函数。

1.2 三叉树利率模型

类似于使用二叉树来描述股价变化过程，我们可以用三叉树描述瞬时无风险利率 $r(t)$ 的变化过程。在使用三叉树时，当利率过大或过小时，我们可以调整树形上节点的分叉方式来保证向分叉各方向变化的概率均非负，并保持使用树形的精度。而且由于描述股价变化的树形中股价是成比例上升或下降，不会出现负股价节点。但描述利率变化的树形中利率是按固定的数值上升或者下降，如果利率过小时不改变树形分叉方式则将会出现过多无效的负利率节点。

这里不过多重复《期权、期货及其他衍生产品》书中内容 [1, chapter 31]。但简单说明几点：

1. 树形中每个节点处的利率 R 实际为从该节点的时刻开始，之后 Δt 时间段内的短期无风险利率，等于 $r(t)$ 在该段时间的平均值，我们近似认为 R 的变化过程和 $r(t)$ 一致。而且对于 $steps$ 次分叉的树形将会有 $steps+1$ 个 Δt 时间段，这和股价树形 $steps$ 次分叉有 $steps$ 个 Δt 不同。

2. 树形的生成主要分为两个部分：(a) 忽略模型中的 $\theta(t)$ 项，考虑树形为一对称结构。(b) 通过要求由树形计算出的 $P(0,t_i)$ 和市场上初始利率期限结构相符合来对每层利率进行调整，以实现引入 $\theta(t)$ 项相同的目的，这里的 t_i 为树形上每层对应的时刻。

3. 关于节点中的 Q 值，其代表瞬时无风险利率从根节点处开始，到该节点所有可能路径下零息债券价格的加权平均。零息债券价格依赖于利率变化路径，同时权重为该路径上所有分叉概率的乘积。

1.3 零息债券期权

零息债券期权对应标的资产为零息债券，其中债券到期时间 $T2$ 大于期权到期时间 $T1$，债券的本金一般记为 L，期权的执行价格为 K。零息债券的价格和期权价格的贴现都只依赖于瞬时无风险利率的变化过程。当我们假设利率的变化过程服从 Hull-White 单因子模型后，欧式零息债券期权的价格有解析表达式。但这里作为使用利率三叉树形的例子，我们也可以使用利率树形来计算欧式零息债券期权的价格。一种方式是先生成 $r(t)$ 由 $t=0$ 至 $t=T2$ 变化的树形，由债券本金 L 从债券到期日 $T2$ 往回倒推出期权执行时刻债券在每个节点的价格，然后对期权价格使用相应节点处的 Q 值进行加权和贴现。另一种方式是只生成 $r(t)$ 由 $t=0$ 至 $t=T1$ 时的利率树形，在树形末层，根据每个节点处的利率，使用 $P(T1,T2)$ 的解析表达式来直接计算出节点处债券的价格，再用 Q 值进行加权和贴现。

这里我们使用和书上例子中一样的解析和树形并用的方法来计算欧式零息债券期权的价格。

2 生成树形和计算债券期权价格

2.1 生成利率三叉树

1. 确定相关条件。对于 0 至 T 之间 $steps$ 次分叉的三叉树，$\Delta t = T/(steps+1)$, $\Delta R = \sigma\sqrt{3\Delta t}$。考虑节点 (i,j)，i 为层数，$i=0,1,...,steps$，j 为该层上节点的位置，$j=-i,-i+1,...,i-1,i$。当层数 $i \geq j_{max}$ 时，j 的取值范围将固定为 $-j_{max},...,j_{max}$，这里 j_{max} 为大于或等于 $0.184/(a\Delta t)$ 的最小整数。对于 $j=\pm j_{max}$ 的节点，其分叉方式为书中所描述的非标准分叉。

2. 由已知的市场上零息利率期限结构生成树形各层的时刻对应的 $P(0,t_i)$。比如已知初始的零息利率为 $[[T_0,r_0],...,[T_M,r_M]]$ 我们需要计算出 $P(0,t_0=0)$, $P(0,t_1=\Delta t),...,P(0,t_{steps+1}=T)$。计算时未知的利率由已知的利率期限结构线性插值的出。这些债券的价格会在计算树形每层利率的调整时被用到。

3. 初始化树形的一个根节点。树形中的节点在 C 中可以自定义一个 struct 结构。三叉树中节点属性包括：”prob”：到达该节点的概率，”Probs”: [Pd, Pm, Pu]：节点向不同方向分叉的概率，”Q”：到达该节点所有可能路径的概率与贴现的加权，”R”：从该节点开始后 Δt 时间段内的短期无风险利率。对于根节点，其初始化为”prob”: 1, ”Probs”: [Not-Defined, Not-Defined, Not-Defined], ”Q”: 1, ”R”: Not-Defined。

4. 从 $i=0$ 层开始，到第 $i=steps-1$ 层，依次处理该层节点同时初始化下一层的节点。具体为：

   a. 初始化下一层的节点，每个节点为: ”prob”:0, ”Probs”:[Not-Defined, Not-Defined, Not-Defined], ”Q”: 0, ”R”: Not-Defined 。

   b. 由当前第 i 层各节点已知的 Q 值和前面计算出的 $P(0,t_{i+1})$，计算出 α_i，得出节点 (i,j) 处 $R_{ij} = j\Delta R + \alpha_i$，并填入每个节点的”R”。

   c. 将每个节点向不同方向分叉的概率计算出并填入该节点。

   d. 将该层每个节点已知的到达该节点的”Q” 和”prob”，按该节点分叉方向和对应概率传递给下一层相应节点。其中”Q” 的传递需要考虑当前节点利率在 Δt 时间段的贴现值。

5. 树形的最后一层 $i=steps$，由于不会继续分叉，只需要同上计算出该层每个节点处的短期无风险利率 R 并填入即可。

2.2 计算欧式零息债券期权价格

考虑期权到期日为 $T1$，执行价格为 K，相应债券到期日为 $T2$，本金为 L。使用 Hull-White 单因子利率模型，假定参数 a, σ 已校准。我们使用一个分叉 $steps$ 次的三叉树，这里的 $\Delta t = T1/steps$，树形末层对应时间段为 $T1$ 至 $T1 + \Delta t$，即树形覆盖的时间段为 $t = 0$ 至 $t = T1 + \Delta t$。这么设定是为了和 $P(T1, T2$ 的解析表达式相符合。

1. 在 0 至 $T + \Delta t$ 时间段内由上面所述步骤生成一个利率三叉树。
2. 在树形末层由单因子模型下 $P(T1, T2)$ 的解析表达式 [1, chapter 31] 计算出每个节点对应的债券价格。
3. 由债券价格和期权执行价格判断是否执行期权，得到该末层每个节点处期权价格。
4. 计算出末层每个节点处期权价格乘以该节点的 Q 之和，即为最终需要的零息债券期权价格。

3 步骤 C 代码实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct Node
{
    int lvl;
    int j;
    double prob;
    double Probs[3];
    // 节点向不同方向分叉概率 Probs[0]:P_d, Probs[1]:P_m, Probs[2]:P_u.
    double R;
    double Q;
};

struct Rates_tree
{
    int steps;
    double T;
    double a;
    double sigma;
    double ** init_rates;

    struct Node ** tree;
};

double * calculate_init_zero_bonds(double ** init_rates, double T, int steps)
{
    double dt = T/(steps+1.0);
    double * Pti = malloc((steps+2) * sizeof(double));
    int p = 0;
    double ti, ri;
    for (int i=0; i<steps+2; i++)
    {
        ti = dt*i;
        while (ti>init_rates[p][0])
        {
```

```
            p++;
        }
        if (p==0)
        {
            ri = init_rates[0][1];
        }
        else
        {
            ri = init_rates[p-1][1]+(init_rates[p][1]-init_rates[p-1][1])\
                /(init_rates[p][0]-init_rates[p-1][0])*(ti-init_rates[p-1][0]);
        }
        Pti[i] = exp(-ri*ti);
    }
    return Pti;
}

// 生成利率树形函数。
struct Rates_tree * build_rates_tree(double a, double sigma, double T, int steps, double ** init_rates,
    int num_init_rates)
{
    struct Rates_tree * rates_tree = malloc(sizeof(struct Rates_tree));
    // 把相关参数保存在树形结构体内。
    rates_tree->a = a;
    rates_tree->sigma = sigma;
    rates_tree->T = T;
    rates_tree->steps = steps;
    rates_tree->tree = malloc((steps+1)*sizeof(struct Node *));

    rates_tree->init_rates = malloc(num_init_rates*sizeof(double *));
    for (int i=0; i<num_init_rates; i++)
    {
        rates_tree->init_rates[i] = malloc(2*sizeof(double));
        rates_tree->init_rates[i][0] = init_rates[i][0];
        rates_tree->init_rates[i][1] = init_rates[i][1];
    }

    // 构建树形的参数。
    double dt = T/(steps+1.0);
    double dR = sigma*sqrt(3.0*dt);
    int j_max = (int)ceil(0.184/a/dt);
    double * Pti = calculate_init_zero_bonds(rates_tree->init_rates, T, steps);

    // 用较短的名称替代长的名字。
    struct Node ** tree = rates_tree->tree;

    tree[0] = malloc(1*sizeof(struct Node));
    tree[0][0].lvl = 0;
    tree[0][0].j = 0;
    tree[0][0].prob = 1.0;
    tree[0][0].Q = 1.0;
    tree[0][0].R = -1.E100;
    tree[0][0].Probs[0] = -1.E100;
```

```
tree[0][0].Probs[1] = -1.E100;
tree[0][0].Probs[2] = -1.E100;

int N1 = steps<j_max ? steps : j_max; // Number of the first standard branching levels.

// 一个辅助指针。
struct Node * node;

double alpha, S;
// 首先构建非标准分叉之前的树形。
for (int lvl=0; lvl<N1; lvl++)
{
    // 生成下一层的空节点。
    tree[lvl+1] = malloc((2*lvl+3)*sizeof(struct Node));
    for (int j=-lvl-1; j<lvl+2; j++)
    {
        node = &tree[lvl+1][j+lvl+1];
        node->lvl = lvl+1;
        node->j = j;
        node->prob = 0;
        node->Q = 0;
        node->R = -1.E100;
        node->Probs[0] = -1.E100;
        node->Probs[1] = -1.E100;
        node->Probs[2] = -1.E100;
    }

    // 计算和处理该层的节点。
    // 1. 计算 R = dR*j+alpha.
    // 2. 计算各个方向分叉的概率。
    // 3. 向下一层的节点传递概率和Q值。
    S = 0;
    for (int j=-lvl; j<lvl+1; j++)
    {
        S += tree[lvl][j+lvl].Q * exp(-j*dR*dt);
    }
    alpha = (log(S)-log(Pti[lvl+1]))/dt;

    for (int j=-lvl; j<lvl+1; j++)
    {
        node = &tree[lvl][j+lvl];
        node->R = j*dR+alpha;
        node->Probs[0] = 1.0/6.0+0.5*(a*a*j*j*dt*dt+a*j*dt);
        node->Probs[1] = 2.0/3.0-a*a*j*j*dt*dt;
        node->Probs[2] = 1.0/6.0+0.5*(a*a*j*j*dt*dt-a*j*dt);

        for (int k=0; k<3; k++)
        {
            tree[lvl+1][j+lvl+k].prob += node->prob * node->Probs[k];
            tree[lvl+1][j+lvl+k].Q += node->Q * node->Probs[k] * exp(-node->R*dt);
        }
    }
```

```
}

// 判断该利率树中是否会有非标准分叉的层。
// 如有非标准分叉，先处理出现非标准分叉层中标准分叉节点，
// 再处理上下边界处的非标准分叉节点。
if (steps<=j_max)
{}
else
{
   for (int lvl=j_max; lvl<steps; lvl++)
   {
      tree[lvl+1] = malloc((2*j_max+1)*sizeof(struct Node));
      for (int j=-j_max; j<j_max+1; j++)
      {
         node = &tree[lvl+1][j+j_max];
         node->lvl = lvl+1;
         node->j = j;
         node->prob = 0.0;
         node->Q = 0.0;
         node->R = -1.E100;
         node->Probs[0] = -1.E100;
         node->Probs[1] = -1.E100;
         node->Probs[2] = -1.E100;
      }

      S = 0;
      for (int j=-j_max; j<j_max+1; j++)
      {
         S += tree[lvl][j+j_max].Q*exp(-j*dR*dt);
      }
      alpha = (log(S)-log(Pti[lvl+1]))/dt;

      // 先处理中间部分标准分叉节点。
      for (int j=-j_max+1; j<j_max; j++)
      {
         node = &tree[lvl][j+j_max];

         node->R = j*dR+alpha;
         node->Probs[0] = 1.0/6.0+0.5*(a*a*j*j*dt*dt+a*j*dt);
         node->Probs[1] = 2.0/3.0-a*a*j*j*dt*dt;
         node->Probs[2] = 1.0/6.0+0.5*(a*a*j*j*dt*dt-a*j*dt);

         for (int k=0; k<3; k++)
         {
            tree[lvl+1][j+j_max+k-1].prob += node->prob * node->Probs[k];
            tree[lvl+1][j+j_max+k-1].Q += node->Q * node->Probs[k] * exp(-node->R*dt);
         }
      }
      // 处理上下边界处非标准分叉节点。
      int j = -j_max;
      node = &tree[lvl][0];
      node->R = j*dR+alpha;
```

```
            node->Probs[0] = 7.0/6.0+0.5*(a*a*j*j*dt*dt+3.0*a*j*dt);
            node->Probs[1] = -1.0/3.0-a*a*j*j*dt*dt-2.0*a*j*dt;
            node->Probs[2] = 1.0/6.0+0.5*(a*a*j*j*dt*dt+a*j*dt);
            for (int k=0; k<3; k++)
            {
                tree[lvl+1][k].prob += node->prob * node->Probs[k];
                tree[lvl+1][k].Q += node->Q * node->Probs[k] * exp(-node->R*dt);
            }
            j = j_max;
            node = &tree[lvl][2*j_max];
            node->R = j*dR+alpha;
            node->Probs[0] = 1.0/6.0+0.5*(a*a*j*j*dt*dt-a*j*dt);
            node->Probs[1] = -1.0/3.0-a*a*j*j*dt*dt+2.0*a*j*dt;
            node->Probs[2] = 7.0/6.0+0.5*(a*a*j*j*dt*dt-3.0*a*j*dt);
            for (int k=0; k<3; k++)
            {
                tree[lvl+1][2*j_max-2+k].prob += node->prob * node->Probs[k];
                tree[lvl+1][2*j_max-2+k].Q += node->Q * node->Probs[k] * exp(-node->R*dt);
            }
        }
    }

    // 计算并填入上面没有处理的树形最后一层。
    S = 0;
    int radius = steps<j_max ? steps : j_max;
    for (int j=-radius; j<radius+1; j++)
    {
        S += tree[steps][j+radius].Q * exp(-j*dR*dt);
    }
    alpha = (log(S)-log(Pti[steps+1]))/dt;
    for (int j=-radius; j<radius+1; j++)
    {
        tree[steps][j+radius].R = j*dR+alpha;
    }

    free(Pti);

    return rates_tree;
}

void free_rates_tree(struct Rates_tree * rates_tree, int num_init_rates)
{
    for (int i=0; i<rates_tree->steps+1; i++)
    {
        free(rates_tree->tree[i]);
    }
    for (int i=0; i<num_init_rates; i++)
    {
        free(rates_tree->init_rates[i]);
    }
    free(rates_tree->tree);
    free(rates_tree->init_rates);
```

```
    free(rates_tree);
}

double P(double t, double ** init_rates)
{
    double r;
    int p = 0;
    while (t>init_rates[p][0])
    {
        p++;
    }
    if (p==0)
    {
        r = init_rates[0][1];
    }
    else
    {
        r = init_rates[p-1][1]+(init_rates[p][1]-init_rates[p-1][1])\
            /(init_rates[p][0]-init_rates[p-1][0])*(t-init_rates[p-1][0]);
    }
    return exp(-r*t);
}

double B(double t, double T, double a)
{
    return (1.0-exp(-a*(T-t)))/a;
}

double PtT_explicit(double t, double T, double R, double dt, double a, double sigma, double **
    init_rates)
{
    double A_hat = P(T, init_rates)/P(t, init_rates);
    A_hat *= exp(-log(P(t+dt, init_rates)/P(t, init_rates))*B(t, T, a)/B(t, t+dt, a));
    A_hat /= exp(sigma*sigma/4.0/a*(1.0-exp(-2.0*a*t))*B(t, T, a)*(B(t, T, a)-B(t, t+dt, a)));

    return A_hat * exp(-B(t, T, a)/B(t, t+dt, a)*dt*R);
}

// 计算欧式零息债券期权函数
double * bond_option(double K, double L, double T1, double T2, int steps, double a,
                     double sigma, double ** init_rates, int num_init_rates)
{
    double * call_put_prices = malloc(2*sizeof(double));
    call_put_prices[0] = 0.0;
    call_put_prices[1] = 0.0;

    double dt = T1/steps;
    struct Rates_tree * rates_tree = build_rates_tree(a, sigma, T1+dt, steps, init_rates,
        num_init_rates);
    struct Node ** tree = rates_tree->tree;

    double bond;
```

```
    int width = 1-2*tree[steps][0].j;
    for (int j=0; j<width; j++)
    {
        bond = L * PtT_explicit(T1, T2, tree[steps][j].R, dt, a, sigma, init_rates);
        if (bond>K)
        {
            call_put_prices[0] += (bond-K) * tree[steps][j].Q;
        }
        else
        {
            call_put_prices[1] += (K-bond) * tree[steps][j].Q;
        }
    }

    free_rates_tree(rates_tree, num_init_rates);

    return call_put_prices;
}

void print_tree(struct Rates_tree * rates_tree)
{
    printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
    for (int i=0; i<rates_tree->steps+1; i++)
    {
        printf("\ni = %d\n R: ", i);
        for (int j=0; j<1-2*rates_tree->tree[i][0].j; j++)
        {
            printf("%.4lf ", rates_tree->tree[i][j].R);
        }
        printf("\n Q: ");
        for (int j=0; j<1-2*rates_tree->tree[i][0].j; j++)
        {
            printf("%.4lf ", rates_tree->tree[i][j].Q);
        }
    }
    printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
}
```

4 计算示例

我们参考《期权、期货及其他衍生产品》书中第 31 章例 31-4，债券期权为零息债券上的欧式看跌期权，期权执行时间为 $T1 = 3.0$，债券到期时间为 $T2 = 9.0$，期权执行价为 $K = 63.0$，债券本金为 $L = 100.0$。初始利率期限结构已知，单因子模型中已知参数 $a = 0.1, \sigma = 0.01$。

然后我们可以按前面所述步骤生成一个利率三叉树，并计算出期权的价格。对于 50 步的树形，结果为 1.80934；100 步的树形，结果为 1.81444；200 步的树形，结果为 1.80974；500 步的树形，结果为 1.80928 。这些结果和纯解析计算出的该期权价格 1.8093 比较相符。

```
int main()
{
    int num_init_rates = 15;
    double rates[15][2] = {{3/365.0, 0.0501722}, {31/365.0, 0.0498284}, {62/365.0, 0.0497234},\
```

```
                        {94/365.0, 0.0496157}, {185/365.0, 0.0499058}, {367/365.0, 0.0509389},\
                        {731/365.0, 0.0579733}, {1096/365.0, 0.0630595}, {1461/365.0, 0.0673464},\
                        {1826/365.0, 0.0694816}, {2194/365.0, 0.0708807}, {2558/365.0, 0.0727527}, \
                      {2922/365.0, 0.0730852}, {3287/365.0, 0.0739790}, {3653/365.0, 0.0749015}};
    double ** init_rates = malloc(15*sizeof(double*));
    for (int i=0; i<num_init_rates; i++)
    {
        init_rates[i] = malloc(2*sizeof(double));
        init_rates[i][0] = rates[i][0];
        init_rates[i][1] = rates[i][1];
    }

    double a = 0.1;
    double sigma = 0.01;
    double t = 3.0;
    double T = 9.0;
    double K = 63.0;
    double L = 100.0;
    int steps = 200;

    double * call_put_prices = bond_option(K, L, t, T, steps, a, sigma, init_rates, num_init_rates);

    printf("Interest rate tree branching steps: 200 .\n");
    printf("Call price: %.5f , Put price: %.5f .\n", call_put_prices[0], call_put_prices[1]);

    for (int i=0; i<num_init_rates; i++)
    {
        free(init_rates[i]);
    }
    free(init_rates);
    free(call_put_prices);
}

Output:
Interest rate tree branching steps: 200 .
Call price: 1.05458 , Put price: 1.80974 .
```

5 参考资料

参考文献

[1]《期权、期货及其他衍生产品》（原书第 9 版）第 31 章，John C. Hull 著，王勇、索吾林译，机械工业出版社，2014.11 。