时间按变化步长推进

变化步长最主要的思想就是，从仿真一开始就通过从分布取样(sample from the distribution)，确定各个元件的使用寿命，从而将仿真过程的内层循环次数减少。

具体而言，就是在时间轴上首先取样(sample)出切换器A、B的故障发生时间，充分利用故障只能发生一次的特点，每次内循环只需要按照发生故障的先后顺序更新节点的状态即可。

因固定步长的代码略去不讲，这里不再详细对比二者的区别，仅在必要的时候引用固定步长代码以帮助理解。

```
md"## 时间按变化步长推进
变化步长最主要的思想就是,从仿真一开始就通过从分布取样(sample from the distribution),确定各个元件的使用寿命,从而将仿真过程的内层循环次数减少。

具体而言,就是在时间轴上首先取样(sample)出切换器A、B的故障发生时间,充分利用故障只能发生一次的特点,每次内循环只需要按照发生故障的先后顺序更新节点的状态即可。

因固定步长的代码略去不讲,这里不再详细对比二者的区别,仅在必要的时候引用固定步长代码以帮助理
解。
"
```

案例2代码实现

本部分将详细解释各部分代码的具体含义

```
md"## 案例2代码实现
本部分将详细解释各部分代码的具体含义
"
```

注意：在 `julia` 语言中，`begin-end` 语句并不会引入一个新的命名空间。(命名空间为namespace，但原本的用词是scope block，这里为方便理解和查询相关词汇含义而改写用词，有不恰当之处)

```
md"**注意**:在`julia`语言中, `begin-end`语句并不会引入一个新的命名空间.(命名空间为namespace,但原本的用词是scope block,这里为方便理解和查询相关词汇含义而改写用词,有不恰当之处)"
```

导入标准库

各个标准库主要使用到的函数已经列在行末注释，无需特别关注本部分，函数名均较为直观，容易理解。

```
md"### 导入标准库
各个标准库主要使用到的函数已经列在行末注释,无需特别关注本部分,函数名均较为直观,容易理解。"
```

```julia
begin
    using Statistics          # to use mean()
    using Printf              # to use @printf()
    using Plots               # to use histogram() and histogram!()
    using DelimitedFiles      # to use writedlm() (and readdlm())
    using Distributions       # to use Exponential() distribution for sampling
    using PlutoUI             # to use @with_terminal
    using Random              # to use rand!() inplace operation
end
```

常量参数

本部分为所需的常量参数，命名与指导书中一致。

注意：使用了部分语言不支持的Unicode变量名λA,λB。

```julia
md"### 常量参数
本部分为所需的常量参数,命名与指导书中一致。

**注意**:使用了部分语言不支持的Unicode变量名λA,λB。"
```

```julia
begin
    # constant parameters
    ## system
    const NUM_SYSTEM = 300_0000
    # const NUM_SYSTEM = 10_0000
    const NUM_NODE = 10             # this should also be optimized later
    const TIME_STEP = 1             # 1 hour
    const LIFE_LIMIT = 20_0000
    # const LIFE_LIMIT = 20_0000
    const STATE_NUM_NODE = 6
    const k = 3
    ## switch A
    const λA = 1 / 5.90e4           # hour
    const PA0 = exp(-λA * TIME_STEP)
    const PEA1 = 0.20 * (1 - PA0)
    const PEA2 = 0.15 * (1 - PA0)
    const PEA3 = 0.65 * (1 - PA0)
    ## switch B
    const λB = 1 / 2.20e5           # hour
    const PB0 = exp(-λB * TIME_STEP)
    const PEB1 = 0.45 * (1 - PB0)
    const PEB2 = 0.55 * (1 - PB0)
    nothing
end
```

函数定义

函数末尾带有`!`的含义是该函数将参数的引用传入内部，函数内部修改参数值会直接作用到原变量，以及不拷贝参数副本。这样做是为了提高运行效率。

注意：非常不建议按顺序逐个阅读函数，推荐的方式是打开2份本文档，跟随`julia_main_varia()`和`simulate_variable_timestep!()`的代码执行过程，查看对应函数的具体实现。

```
md"### 函数定义
函数末尾带有`!`的含义是该函数将参数的引用传入内部，函数内部修改参数值会直接作用到原变量，以及不拷贝参数副本。这样做是为了提高运行效率。

**注意**：非常不建议按顺序逐个阅读函数，推荐的方式是打开2份本文档，跟随`julia_main_varia()`和`simulate_variable_timestep!()`的代码执行过程，查看对应函数的具体实现。"
```

初始化运行

本函数清空上一轮的仿真状态，用于新一轮仿真的启动

```
md"#### 初始化运行
本函数清空上一轮的仿真状态，用于新一轮仿真的启动"
```

initialize! (generic function with 1 method)

```
function initialize!(gA, gB, gN)
    fill!(gA, 0)  # 状态全清0,代表正常
    fill!(gB, 0)
    # 节点状态均为完好
    fill!(gN, 0)
    # 随机选取1个节点为主节点
    master_node::Int8 = rand(1:10)
    return master_node
end

```

estimate_switch_state! (generic function with 1 method)

```
function estimate_switch_state!(gA, gB, lifeA, lifeB)
    # 这里相当于是假定,所有的switch必然会失效
    distrA = Exponential(1 / λA)
    distrB = Exponential(1 / λB)
    rand!(distrA, lifeA)  # 生成服从指数分布的随机数
    rand!(distrB, lifeB)
    @inbounds for i = 1:NUM_NODE
        tolA = rand() * (1 - PA0)
        gA[i] = tolA < PEA1 ? 1 : tolA < PEA1 + PEA2 ? 2 : 3
        tolB = rand() * (1 - PB0)
        gB[i] = tolB < PEB1 ? 1 : 2
    end
    nothing
end
```

```
estimate_node_state! (generic function with 1 method)
```

```julia
function estimate_node_state!(gA, gB, gN, lifeA, lifeB, switch_tag, idx)
    if switch_tag
        # 刚刚坏掉的是switchA[idx],需要重新计算当前的节点状态
        if gA[idx] == 0  # 这条语句不会被执行
        elseif gA[idx] == 1
            lifeB[idx] != +Inf && (gN[idx] = 1; return nothing)
            gB[idx] == 1 && (gN[idx] = 5; return nothing)
            gB[idx] == 2 && (gN[idx] = 1; return nothing)
        elseif gA[idx] == 2
            lifeB[idx] != +Inf && (gN[idx] = 2; return nothing)
            gB[idx] == 1 && (gN[idx] = 3; return nothing)
            gB[idx] == 2 && (gN[idx] = 4; return nothing)
        elseif gA[idx] == 3
            lifeB[idx] != +Inf && (gN[idx] = 4; return nothing)
            gB[idx] == 1 && (gN[idx] = 4; return nothing)
            gB[idx] == 2 && (gN[idx] = 4; return nothing)
        end
    else
        if gB[idx] == 0
            # 什么也不做,因为这条语句不可能被执行
        elseif gB[idx] == 1
            lifeA[idx] != +Inf && (gN[idx] == 3; return nothing)
            gA[idx] == 1 && (gN[idx] = 5; return nothing)
            gA[idx] == 2 && (gN[idx] = 3; return nothing)
            gA[idx] == 3 && (gN[idx] = 4; return nothing)
        elseif gB[idx] == 2
            lifeA[idx] != +Inf && (gN[idx] == 1; return nothing)
            gA[idx] == 1 && (gN[idx] = 1; return nothing)
            gA[idx] == 2 && (gN[idx] = 4; return nothing)
            gA[idx] == 3 && (gN[idx] = 4; return nothing)
        end
    end
    nothing
end
```

```
ok_for_master (generic function with 1 method)
```

```julia
function ok_for_master(gNi)
    gNi == 0 && return true
    gNi == 1 && return false
    gNi == 2 && return true
    gNi == 3 && return true # MO
    gNi == 4 && return false
    gNi == 5 && return false
end
```

```
reselect_master! (generic function with 1 method)
```

```julia
# 节点重选
function reselect_master!(gN, master_node)  # 1
    if !ok_for_master(gN[master_node])
        # 说明主节点坏掉了
        alert_mintime = 1.0 # rand() [0.0,1)
        mintime_index = 0
        alert_count = rand(NUM_NODE)
        @inbounds for i = 1:NUM_NODE
            if ok_for_master(gN[i])
                tmp = alert_mintime
                alert_mintime = min(alert_mintime, alert_count[i])
                mintime_index = tmp != alert_mintime ? i : mintime_index
            end
        end
        mintime_index == 0 && return nothing
        master_node = mintime_index
    end
    # 如果原来的主节点可用,通常情况下是不需要进行节点重选的,
    # 这个时候需要把需要进行节点重选的情况筛选出来
    # 信号出现异常,这个时候要么是MO,要么是FB
    # 我们只需要讨论一种情况,如果系统有一个MO出现
    cnt = 0
    idx = 0
    flag = false # flag for FB
    @inbounds for i = 1:NUM_NODE
        gN[i] == 3 && (cnt = cnt + 1; idx = i; continue)
        gN[i] == 5 && (flag = true; continue)
    end
    if cnt == 1
        master_node = idx
        # elseif cnt >= 2
        #     master_node = 0 # 这里随意设置,因为系统必然失效
        # elseif cnt == 0 && flag
        #     master_node = 0 # 这里随意设置,因为系统必然失效
        # else
        #     # 也就是说现在信号是好的,不需要重选
    end
    nothing
end
```

```
estimate_system_state! (generic function with 1 method)
```

```
function estimate_system_state!(gN,master_node)
    QPF::Int8 = QSO::Int8 = QDM::Int8 = QMO::Int8 = QDN::Int8 = QFB::Int8 = 0
    Gsys::Int8 = 2 # wtf 这里见鬼了,Gsys存在没有覆盖到的状态
    @inbounds for elem in gN
        elem == 0 && (QPF += 1; continue)
        elem == 1 && (QSO += 1; continue)
        elem == 2 && (QDM += 1; continue)
        elem == 3 && (QMO += 1; continue)
        elem == 4 && (QDN += 1; continue)
        elem == 5 && (QFB += 1; continue)
    end
    if QFB >= 1 || QMO >= 2 || QPF + QMO + QDM == 0 || (QPF + QSO + 1((QMO + QDM) >
0)) < k
        Gsys = 1
    elseif QFB == 0 && ((QMO == 1 && QPF + QSO >= k - 1) || ((QMO == 0 && QPF >= 1
&& QPF + QSO >= k) || (QMO == 0 && QPF == 0 && QDM >= 1 && QSO >= k - 1)))
        Gsys = 2
        # 这里的条件文本没有给清楚,但大致能猜到C5 && (C6 || C7)
    elseif QFB + QMO == 0 && (QPF >= 1 && QPF + QSO == k - 1 && QDM >= 1)
        if gN[master_node] == 2
            # if one of gDM is selected as master
            Gsys = 3
        elseif gN[master_node] == 0
            # if one of gPF is selected as master
            # fail to satisfy valid node limit k
            Gsys = 4
        end
    end
    return Gsys
end
```

仿真执行

单次仿真的完整执行过程

- `initialize!()`
- `estimate_switch_state!()`
- loop: iter switches, select current `min_life`
    - find `min_life`, find its `index` and switch type `X`
    - if `min_life` > `LIFE_LIMIT`, break loop, stop simulation, return previous `life_counter`
    - else, `estimate_node_state!()`
    - set corresponding `lifeX[index]` to `+Inf`, avoid repeat selection in the following simulation iterations
    - `reselect_master!()`
    - `estimate_system_state!()`
    - update `life_counter`
- return `life_counter`

```
md"#### 仿真执行
单次仿真的完整执行过程
- `initialize!()`
- `estimate_switch_state!()`
- loop: iter switches, select current `min_life`
  - find `min_life`, find its `index` and switch type `X`
  - if `min_life` > `LIFE_LIMIT`, break loop, stop simulation, return previous `life_counter`
  - else, `estimate_node_state!()`
  - set corresponding `lifeX[index]` to `+Inf`, avoid repeat selection in the following simulation iterations
  - `reselect_master!()`
  - `estimate_system_state!()`
  - update `life_counter`
- return `life_counter`
"
```

```
simulate_variable_timestep! (generic function with 1 method)
```

```julia
function simulate_variable_timestep!(gA, gB, gN, lifeA, lifeB)
    master_node::Int8 = initialize!(gA, gB, gN)
    state_system = false
    life_counter::Float64 = 0
    estimate_switch_state!(gA, gB, lifeA, lifeB)
    @inbounds for i = 1:2*NUM_NODE
        minA, idxA = findmin(lifeA)
        minB, idxB = findmin(lifeB)
        min_life = min(minA, minB)

        min_life > LIFE_LIMIT && break  # 筛选到的寿命大于了限定的最大值
        switch_tag, idx = (min_life == minA) ? (true, idxA) : (false, idxB)
        # switch_tag=true => minA, switch_tag=false => minB
        estimate_node_state!(gA, gB, gN, lifeA, lifeB, switch_tag, idx)
        if switch_tag
            lifeA[idxA] = +Inf
        else
            lifeB[idxB] = +Inf
        end
        # start from here, 2 methods for variable time step
        reselect_master!(gN, master_node)
        Gsys::Int8 = estimate_system_state!(gN, master_node)
        if Gsys == 2 || Gsys == 3
            # life_counter = max(min_life, life_counter)
            life_counter = min_life
        else
            # life_counter = max(min_life, life_counter)
            # state_system = true
            break
        end
    end
    # @printf("\ngA:")
    # for i = 1:NUM_NODE
    #     @printf("%3d", gA[i])
    # end
    # @printf("\ngB:")
    # for i = 1:NUM_NODE
    #     @printf("%3d", gB[i])
    # end
    # @printf("\ngN:")
    # for i = 1:NUM_NODE
    #     @printf("%3d", gN[i])
    # end
    # @printf("\nGsys:%3d\tQPF%3d\tQSO%3d\tQDM%3d\tQMO%3d\tQDN%3d\tQFB%3d\n", Gsys, QPF, QSO, QDM, QMO, QDN, QFB)
    life_counter
end
```

julia_main_varia (generic function with 1 method)

```
function julia_main_varia()
    # variables definition for each simulation
    gA = zeros(Int8, NUM_NODE)
    gB = zeros(Int8, NUM_NODE)
    gN = zeros(Int8, NUM_NODE)
    system_life = zeros(Float64, NUM_SYSTEM)
    lifeA = zeros(NUM_NODE)
    lifeB = zeros(NUM_NODE)
    # run simulation
    @time @inbounds for i = 1:NUM_SYSTEM
        # @printf("\nsystem number:%8d\n", i)
        system_life[i] = simulate_variable_timestep!(gA, gB, gN, lifeA, lifeB)
        # @printf("system life:%16.4f\n", system_life[i])
    end
    # bugval = count(i -> (i >= LIFE_LIMIT), system_life)
    # writedlm("out/csv/sys$NUM_SYSTEM-lim$LIFE_LIMIT-variable-has-bugval.csv",
system_life, ',')
    # @printf("Invalid simulations: %8d (%5.2f%%)\n", bugval, bugval * 100.0 /
NUM_SYSTEM)
    # deleteat!(system_life, system_life .>= LIFE_LIMIT)
    # writedlm("out/csv/sys$NUM_SYSTEM-lim$LIFE_LIMIT-variable-no-bugval.csv",
system_life, ',')
    # readdlm("out/csv/FileName.csv",',',Int32)
    @printf("Avg Life: %12.6f\n", mean(system_life))
    # p = histogram(system_life, bins=min(NUM_SYSTEM, 256), label="$(NUM_SYSTEM-
bugval) valid samples")
    p = histogram(system_life, bins=min(NUM_SYSTEM,256), label="$(NUM_SYSTEM)
samples")
    p = histogram!(legend=:topright, bar_edges=true)
    p = histogram!(title="variable Time Step Simulation")
    savefig(p, "out/figure/sys$NUM_SYSTEM-lim$LIFE_LIMIT-variable.png")
end
```

```
  7.480853 seconds (16.95 M allocations: 2.273 GiB, 2.52% gc time)
Avg Life: 104633.564187
```

```
@with_terminal julia_main_varia()
```