信息网络中的随机量建模

引子

- **信息网络中存在很多随机量。**
  - 网络中的业务是随机的；
    - 云计算中的计算任务是随机的；
  - 移动（无线）网络中的节点位置是随机的；
  - 自组织网络中的网络拓扑结构是随机的；
  - 受各种地理条件、建筑、植被等影响，无线传播环境下的信道是随机的；
- **随机量建模对蒙特卡洛网络仿真非常重要。**
  - 输入模型对仿真模型的输出影响很大；
  - 选择适当的分布来表示输入数据是非常关键的；
  - 如何在仿真程序中实现这些随机量模型同样重要。

引子

信息网络仿真中，需要对这些随机量进行准确建模：

建模不是闭门造车，**模型从实测数据中来**；

随机量建模包含两个步骤：

- 从数据到模型：输入建模
  - 如何采集数据？
  - 如何选择正确的分布？
  - 如何估计分布参数？
  - 如何验证选择的正确性？
- 模型在仿真程序中的实现

目录

从数据到模型

- 从真实场景收集数据
- 选择用**非参数**模型还是**参数化概率**模型?

如果选择参数化概率模型

- 1、选择一个适当的**概率分布**来表示输入过程
- 2、估计概率分布**关联参数**
- 3、评估所选分布和关联参数的**拟合度**

从数据到模型

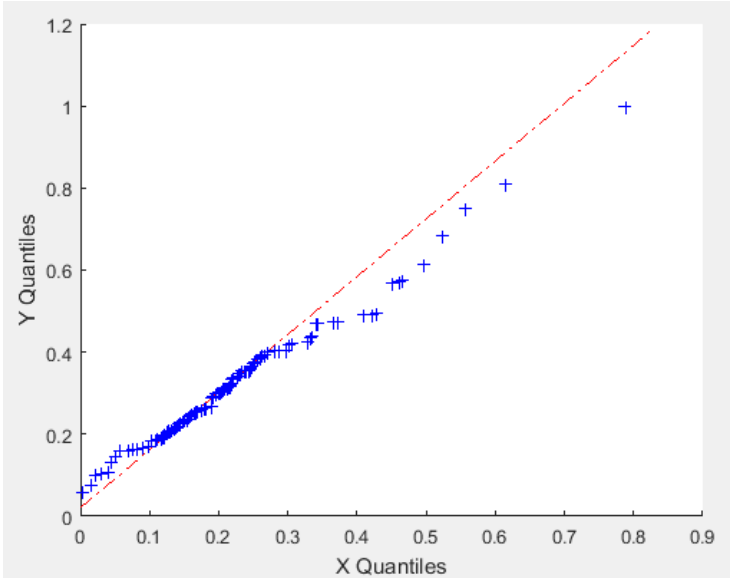
1、数据收集

2、输入建模

- 轨迹驱动仿真（trace-driven simulation）
  - 采集数据直接用于仿真
- 经验输入建模（empirical input modeling）
  - 从采集数据中得到随机变量用于仿真
- 理论输入建模（theoretical input modeling）
  - 理论分布函数的参数根据采集数据进行估计
  - 随机变量由拟合分布函数生成

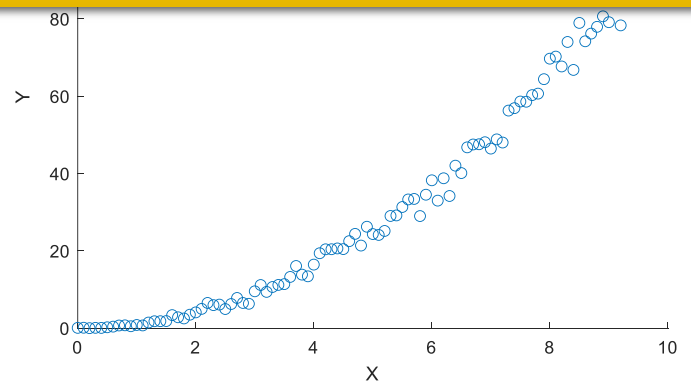
提高数据准确性的建议：

- 对输入过程进行层次划分，对不同层次的数据分别进行收集；
  - 例如HTTP业务建模为：会话级、页面级、连接级、数据包级
- 在收集数据的同时，对所收集数据的有效性进行分析，确保数据可用于信息
- 对
  - 可以用Q-Q图对数据集的同质性进行分析
  - 作用：同质的数据集可以共同建模
- 通过绘制散点图的方式判断两个变量之间是否有关联性；
  - 作用：利用强关联可降低变量数



Q-Q图：横纵坐标分别为两个数据集的分位数

Q-Q图判别样本数据是否近似于正态分布,只需看Q-Q图上的点是否近似地在一条直线附近,而且该直线的斜率为标准差,截距为均值.



散点图

从数据到模型

1、数据收集

2、输入建模

- 轨迹驱动仿真（trace-driven simulation）
  - 采集数据直接用于仿真
- 经验输入建模（empirical input modeling）
  - 从采集数据中得到随机变量用于仿真
- 理论输入建模（theoretical input modeling）
  - 理论分布函数的参数根据采集数据进行估计
  - 随机变量由拟合分布函数生成

经验建模

1、非参数建模（Nonparametric Modeling）

2、个体数据的经验建模（Empirical Modeling of Individual Data）

3、分组数据的经验建模（Empirical Modeling of Grouped Data）

非参数建模

非参数建模：生成的随机输入为从收集数据中以$1/n$的概率进行采样得到的数据，其中n为收集数据的总数。

建模方式：（收集数据为$\{X_k: k = 1 \sim n\}$）

1、生成一个随机均匀分布数$u \sim U(0,1)$;

2、令$P = n \times u$，则$k = [P] + 1$;

3、X_k即为生成的随机输入。

例：假设已收集到某银行ATM机在一个小时内前十名顾客的服务时间数据，如下表所示。如何用非参数建模方式进行建模？

观察数	1	2	3	4	5	6	7	8	9	10
服务时间	56	51	73	65	84	58	62	69	44	66

建模方式：

1、生成一个随机均匀分布数如，$u = 0.369981;$

2、$P = n \times u = 3.69981$，则$k = [P] + 1 = 4;$

3、$X_4 = 65$ 即为生成的随机输入。

经验建模

1、非参数建模（Nonparametric Modeling）

2、个体数据的经验建模（Empirical Modeling of Individual Data）

3、分组数据的经验建模（Empirical Modeling of Grouped Data）

个体数据的经验建模

个体数据的经验建模：将收集到的数据进行递增排列，经验函数为分段线性函数。$F(X_k)=(k-1)/(n-1)$。生成的随机输入x为随机生成的0-1之间的数据在分段函数上对应的值，如图所示。

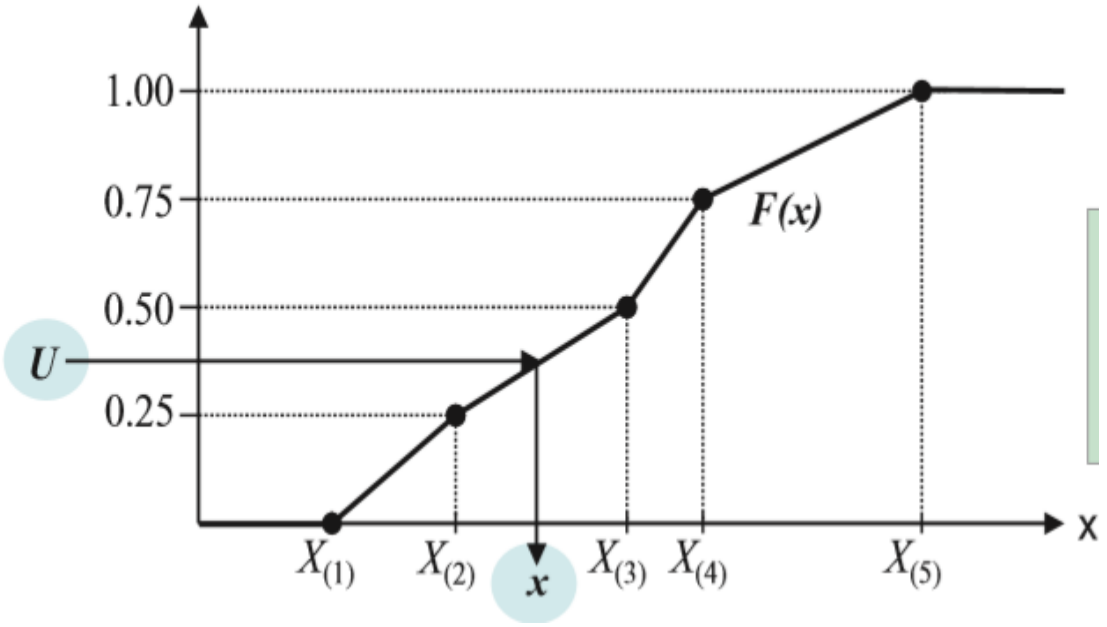
建模方式：（收集数据为$\{X_k: k=1\sim n\}$）

1、对数据以递增方式排列，
然后绘制分段函数$F(X)$;

2、生成一个随机均匀分布数$u\sim U(0,1)$;

3、令$P=(n-1)\times u$，则$J=[P]+1$;

4、$x=X_{(J)}+(P-J+1)\times\left(X_{(J+1)}-X_{(J)}\right)$



个体数据的经验建模

例：假设已收集到某银行ATM机在一个小时内前十名顾客的服务时间数据，如下表所示。如何用个体数据的输入建模方式进行输入建模？

观察数	1	2	3	4	5	6	7	8	9	10
服务时间	56	51	73	65	84	58	62	69	44	66

建模方式：

1、对数据大小进行重新排列,并进行分段函数的绘制；

观察数	1	2	3	4	5	6	7	8	9	10
服务时间	44	51	56	58	62	65	66	69	73	84

2、生成一个随机均匀分布数$u = 0.369981$;

3、$P = (n-1) \times u = 3.328929$，则$\mathrm{J} = [P] + 1 = 4$;

4、$x = X_J + (P - J + 1) \times (X_{J+1} - X_J) = 59.316$即为生成的随机输入。

经验建模

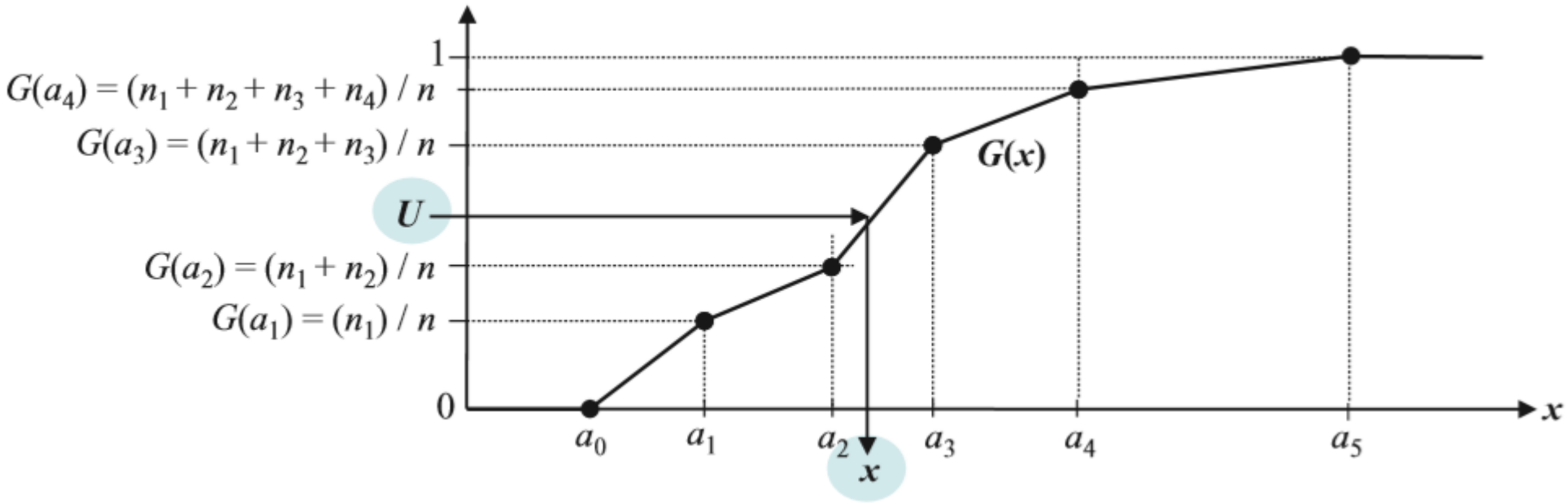
1、非参数建模（Nonparametric Modeling）

2、个体数据的经验建模（Empirical Modeling of Individual Data）

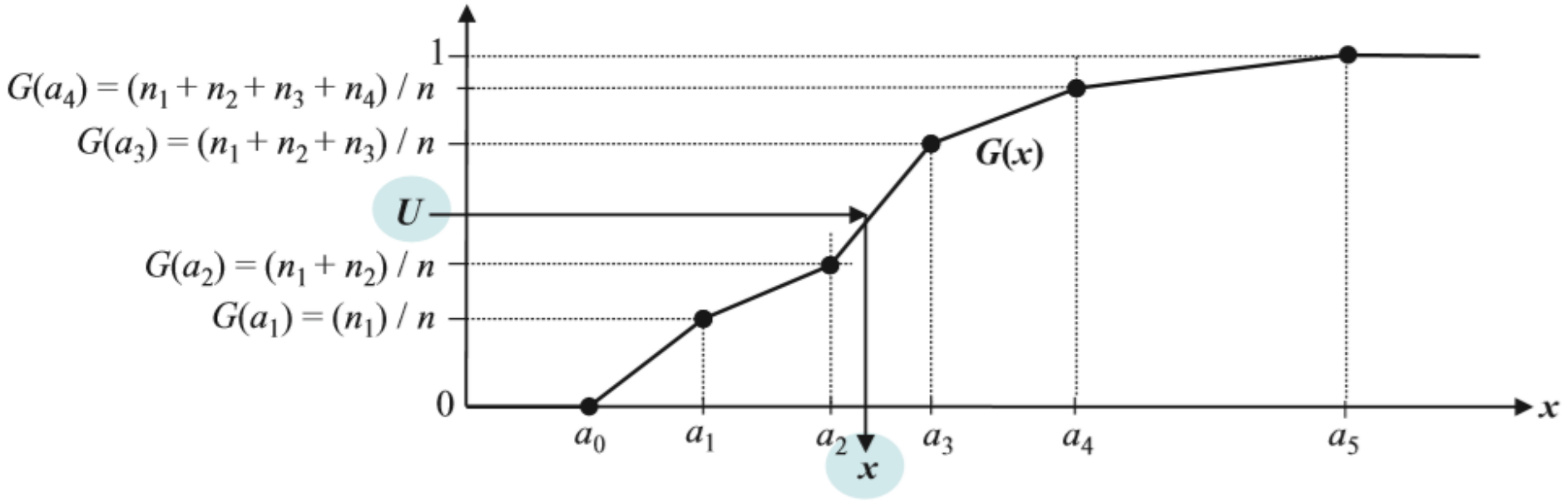
3、分组数据的经验建模（Empirical Modeling of Grouped Data）

分组数据的经验建模：

- 将收集的数据分为m个区间$\{[a_0, a_1), [a_1, a_2), \ldots, [a_{m-1}, a_m)\}$，第$j$个区间包含$n_j$个数据
- 定义经验函数为分段函数：$G(a_0) = 0$，$G(a_j) = \sum_{i=1}^{j} n_i/n$，其中$j = 1 \sim m$，$n = \sum n_j$。
- 生成随机输入x：随机生成的0-1之间的数据在分段函数上对应的值，如图所示（图为m=5的情况）。



分组数据的经验建模



建模方式：（收集数据为$\{X_k: k=1\sim n\}$）

1、以一个合适的区间大小对收集到的数据进行分组$\{[a_0,a_1),[a_1,a_2),\ldots,[a_{m-1},a_m)\}$，并计算每个组所含有的数据数$n_j$，根据数据个数占比绘制经验函数$G(X)$;

2、生成一个随机均匀分布数$u\sim U(0,1)$;

3、使J满足$G(\mathrm{a}_J)\le u\le G(\mathrm{a}_{J+1})$;

4、$x=\mathrm{a}_J+[u-G(\mathrm{a}_J)]\times(\mathrm{a}_{J+1}-\mathrm{a}_J)/[G(\mathrm{a}_{J+1})-G(\mathrm{a}_J)]$。

分组数据的经验建模

例：假设已收集到某银行ATM机在一个小时内前十名顾客的服务时间数据，如下表所示。如何用分组数据的输入建模方式进行输入建模？

观察数	1	2	3	4	5	6	7	8	9	10
服务时间	56	51	73	65	84	58	62	69	44	66

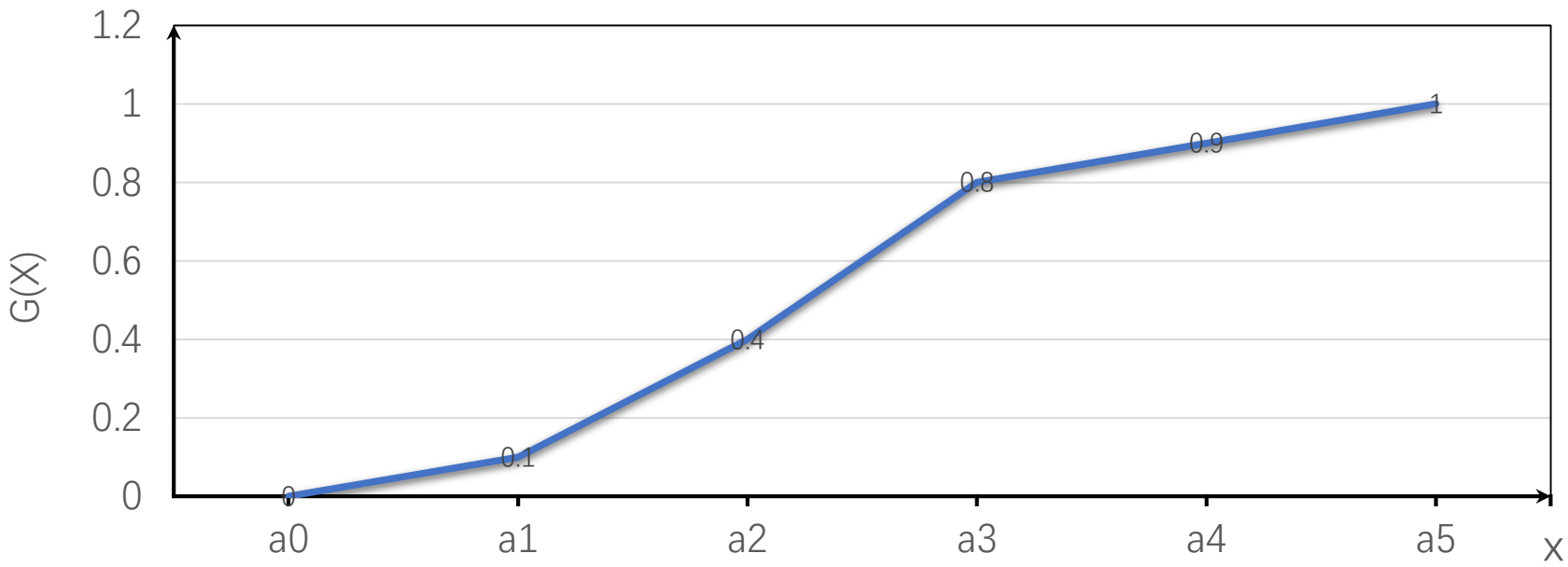
建模方式：

1、对数据大小进行分组，区间大小为10，分为5组并进行分段函数的绘制；

组号	1	2	3	4	5
区间	40-50	50-60	60-70	70-80	80-90
数据数	1	3	4	1	1
G(X)	1/10	4/10	8/10	9/10	10/10

$$a_0 = 40; a_1 = 50; a_2 = 60; a_3 = 70; a_4 = 80; a_5 = 90$$

分组数据的经验建模



2、生成一个随机均匀分布数$u = 0.369981$;

3、使J满足$G(\mathrm{a}_J) \le u \le G(\mathrm{a}_{J+1})$，$J = 1$;

4、$x = \mathrm{a}_J + [u - G(\mathrm{a}_J)] \times \frac{\mathrm{a}_{J+1} - \mathrm{a}_J}{[G(\mathrm{a}_{J+1}) - G(\mathrm{a}_J)]} = 58.9960$。

理论建模

1、数据独立性检查

2、分布函数的选择

3、参数估计

4、拟合度测试

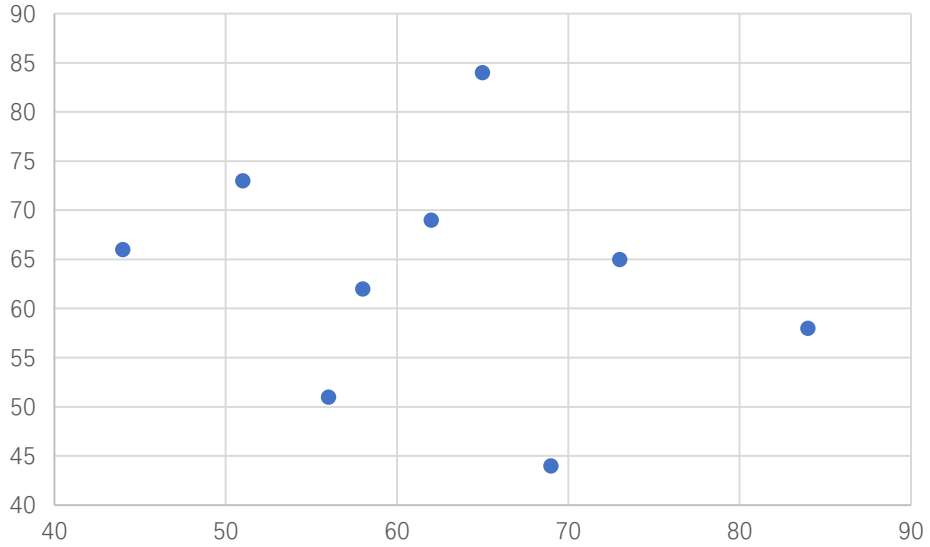
数据独立性检查

简单直观的**独立性**检查方式：

将收集的**一组数据**按照时间收集顺序进行排序，$X_1, X_2, \ldots, X_n$，以X_i为横坐标，X_{i+1}为纵坐标绘制散点图。若散点图为随机分布，则数据独立。

例：假设已收集到某银行ATM机在一个小时内前十名顾客的服务时间数据，如下表所示。判断数据独立性。

观察数	1	2	3	4	5	6	7	8	9	10
服务时间	56	51	73	65	84	58	62	69	44	66



如左图所示，图中数据为随机分布，所以收集到的数据具有独立性。

（这里只考虑了前后两个数据之间的独立性，如果需要，可以考虑X_i和X_{i+k}之间的相关性）

数据独立性检查

- 数据不独立情况下的处理方法：
  - 如果检测到数据不独立，首先应当对所对应的物理过程进行分析，**判断**数据间本身是否就应当具有相关性；
  - 如果是，如下两种建模方法：

如果**数据间本身就应具有相关性**，则应按上一章的随机过程建模方法进行建模；

如果**物理过程可分割为多个不同层次或子过程**，则可以将整个过程划分为层次或子过程，使得划分后的数据统计具有独立性。

理论输入建模

1、数据独立性检查

2、分布函数的选择

3、参数估计

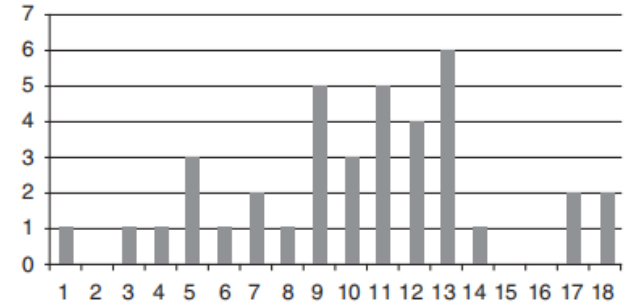
4、拟合度测试

分布函数选择

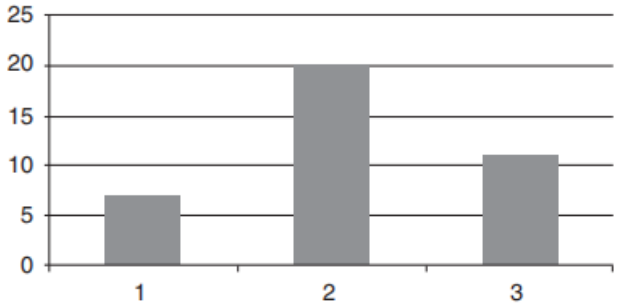
- 分布函数选择：基于理论判断和直方图的形状。
- 在绘制直方图时注意选择合适的直方图区间大小。

一般选择直方图区间数 $=\sqrt{(\text{有效数据数})}$

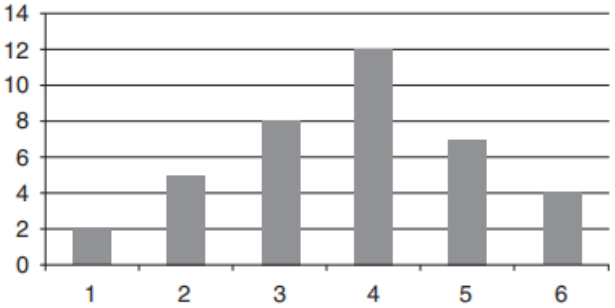
区间过大或过小都无法反映正确的形状



区间过小



区间过大



区间大小适中

分布函数选择

- 选择分布函数时，需考虑数据：
  - 连续 or 离散？

    常见**连续**分布：均匀分布、正态分布、指数分布、t-分布、Gamma分布等

    常见**离散**分布：伯努利分布、泊松分布、二项分布、负二项分布等

  - 是否非负？
  - 是否关于均值对称？

分布函数选择

- 一些特殊的度量值
  - Lexis 比率（Lexis ration）τ：方差与均值之比，用于区分离散分布的类型
    - 数据越**分散**， τ 越大
    - 泊松分布τ =1，二项分布τ<1，负二项分布τ>1
  - 变异系数（Coefficient of variance)：标准差与均值之比
    - 对连续分布常用，如指数分布该值等于1
  - 偏斜：对称性的度量$s = \frac{E(X-EX)^3}{\sigma^3}$
    - 偏斜**接近0**：正态分布、均匀分布等对称分布
    - 偏斜**较大**：指数分布、weibull分布、帕累托分布、瑞利分布等

根据上述数据特征，选择合适的分布函数

理论输入建模

1、数据独立性检查

2、分布函数的选择

3、参数估计

4、拟合度测试

参数估计

最大似然估计：指数分布，正态分布，对数正态分布；

矩量法：Erlang分布，Beta分布。

输入变量类型	分布	参数估计
到达时间间隔	指数分布(θ)	最大似然估计
	Erlang分布(k,θ)	矩量法
服务(修复)时间	Beta分布(α,β)	矩量法
	正态分布(μ,σ)	最大似然估计
	对数正态分布(μ,σ)	最大似然估计

➢ 最大似然估计法：

- 基本思想：利用已知的样本结果信息$x_1, x_2, \ldots, x_n$，反推最具有可能（最大概率）导致这些样本结果出现的模型参数值θ。
- 似然函数：

$$lik(\theta) = f_D(x_1, x_2, \ldots, x_n|\theta)$$

其中，f_D为概率密度函数

- 最大似然求θ：使**似然函数最大**的参数值θ

最大似然举例：正态分布，密度函数为 $N\left(x|\mu,\sigma^2\right)=\frac{1}{\left(2\pi\sigma^2\right)^{\frac{1}{2}}}\exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}$

数据 $\boldsymbol{x}=(x_1,x_2,\ldots,x_N)^T$ 的联合概率为 $p\left(\boldsymbol{x}|\mu,\sigma^2\right)=\prod_{i=1}^{N}N\left(x_i|\mu,\sigma^2\right)$

- 步骤一：将正态分布**代入对数似然函数**

$$\ln L=\ln p\left(\boldsymbol{x}|\mu,\sigma^2\right)=-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2-\frac{N}{2}\ln\sigma^2-\frac{N}{2}\ln(2\pi)$$

- 步骤二：对似然函数求**偏导**

$$\begin{cases}\dfrac{\partial\ln L}{\partial\mu}=\dfrac{1}{\sigma^2}\sum_{i=1}^{N}(x_i-\mu)=0\\[2ex]\dfrac{\partial\ln L}{\partial(\sigma^2)}=-\dfrac{N}{2\sigma^2}+\dfrac{1}{2\sigma^4}\sum_{i=1}^{N}(x_i-\mu)^2=0\end{cases}$$

- 矩量法：

基本思想：计算所采集数据各阶中心矩的统计值**t**，以及所选分布的矩f(**s**)，其中**s**为参数集合。设置 $f(\mathbf{s}) = \mathbf{t}$，通过解方程得到相应参数**s**。

矩量法举例：爱尔兰分布，密度函数为$f(x) = \frac{\theta^{-k}x^{k-1}e^{-x/\theta}}{(k-1)!}$。

- 步骤一：计算**所采集数据各阶中心矩**的统计值。

一阶样本矩：$m_1 = \frac{1}{n}\sum_{i=1}^{n} x_i$；二阶样本矩：$m_2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2$

- 步骤二：**根据密度函数**得到一阶和二阶原点矩。

$$E[X] = \int xf(x)dx = k\theta; \quad E[X^2] = \int x^2 f(x)dx = k(k+1)\theta^2$$

- 步骤三：**建立等式**求解

$$\hat{k} = \frac{(m_1)^2}{[m_2 - (m_1)^2]}; \quad \hat{\theta} = \frac{[m_2 - (m_1)^2]}{m_1};$$

理论输入建模

1、数据独立性检查

2、分布函数的选择

3、参数估计

4、拟合度测试

拟合度测试

- 拟合度测试方法有图形法和解析法两种
- 图形法：
  - P-P图：数据CDF和拟合分布CDF的对应点分别作为横纵坐标
  - Q-Q图：数据分位点和拟合分布分位点分别作为横纵坐标
- 解析法：
  - 卡方检验
  - 柯尔莫可洛夫-斯米洛夫检验（Kolmogorov–Smirnov test，K-S test）

- P-P图是根据变量的累积比例与指定分布的累积比例之间的关系所绘制的图形。通过P-P图可以检验数据是否符合指定的分布。当数据符合指定分布时，P-P图中各点近似呈一条直线。
- 如果在绘制中P-P图中各点不呈直线，但有一定规律，可以对变量数据进行转换，使转换后的数据更接近指定分布。
- P-P图和Q-Q图的用途完全相同，只是检验方法存在差异

例如：Q-Q图判别样本数据是否近似于正态分布,只需看Q-Q图上的点是否近似地在一条直线附近,而且该直线的斜率为标准差,截距为均值.

拟合度测试方法：卡方检验

- 卡方检验方式：将采集数据分为m组$\{[a_0,a_1),[a_1,a_2),\dots,[a_{m-1},a_m)\}$，第$j$个区间包含$n_j$个数据，检验统计量：

$$\chi^2=\sum_{j=1}^{m}\frac{(n_j-np_j)^2}{n_j},\quad p_j=\int_{a_{j-1}}^{a_j}\hat{f}(x)dx$$

其中n为有效数据量总数，$\hat{f}(x)$为拟合分布的密度函数。

- 在计算完检验统计量χ^2后，与（m-1）自由度的卡方值进行检验。
- 卡方检验条件：**n足够大，np_j不能太小**。一般要求**$n\geq 50$**以及**$np_j\geq 5$**。当np_j不足够大时，可以适当将某些区间合并来满足这个条件。

目录

业务源建模简介

- **业务源模型重要性**：在信息网络建模中，业务源的模型至关重要，准确的业务模型能够对业务中的关键特征进行表示。
- 不同的实际应用将产生不同的业务源类型，例如：**http业务、P2P业务、话音业务、视频业务和计算业务**等。
- 建模过程通常包括两步：
  - 首先：通过适当的方法来测量关键特征，方法如：服务器日志、客户端日志、数据包跟踪等方法；
  - 然后：在模型中表示出这些特征，例如采用马尔可夫链等方法。

业务源模型的共性特征

- **层次性**：基本都将业务过程分为多个不同层次，分层建模
- **多参数**：
  - 基本都不是单参数模型
  - 通过对时间间隔、持续时间、数据量大小等多个量分别建模，最终达到模拟整个随机过程的目的
- **协议流程分析**：
  - 业务源模型往往与应用层协议密切相关，这些协议的处理流程基本都遵循一定标准，通过对标准所规定的协议流程的分析，从中划分出**确知量**和真正的**随机量**
  - 直接对这些随机量进行建模，对确知量则直接通过模拟流程得到。

HTTP业务

1.1.1、HTTP业务模型

1.1.2、HTTP参数

1.1.3、NS3中实现HTTP业务

HTTP业务

- WWW业务：因特网的重要部分
- WWW业务**基本概念**：
  - 网页：网络业务建模的重要模块
    - 包括ASCII形式的超文本标记语言（HTML）
    - HTML：定义网页的结构和互联关系
  - 主对象：组成网页页面的主体框架，即一个由HTML描述的文档
  - 嵌入对象：内嵌在页面框架中的文本、声音或者图片等。

```
<object width=“400” height=“400” data=“helloworld.swf”></object>
: 嵌入 一段 .swf
```

  - 下载网页的过程：向网页的URL链接发送请求，通过**HTTP协议**，与网页服务器之间建立TCP连接

HTTP业务

HTTP业务模型

首先需要收集重要的数据来进行分析，以确定适当的参数

然后进行建模

数据采集方法

- **服务器日志：**
  - 服务器跟踪所服务的文件
  - 没有考虑用户行为
- **客户端日志：**
  - 在客户端进行文件跟踪，需要修改浏览器，目前一般不可用
- **数据包追踪：**
  - **最流行的方法**
  - 如：Wireshark、TCP-Dump等工具

建模方法

- **面向页面的模型**

面向页面的建模

面向页面的模型通常以**多层结构**来进行描述，包括会话级、页面级、连接级、数据包级。

- **会话级**：
  - 描述用户的行为
  - 创建一个页面后执行的跳转、重定向、整合等都可以是一个会话，只要不关闭浏览器页面，就是一次会话
  - 在会话中用户浏览多个网页，会话表示为页面的集合
  - 参数：
    - 例如：每个周期的网络会话个数、该周期中的会话分布
- **页面级**：
  - 描述每个会话的网页
  - 参数
    - 例如：每个会话的网页数，两个网页之间时间的分布
    - 每个页面之间的时间建模为阅读时间

面向页面的建模

面向页面的模型通常以**多层结构**来进行描述，包括会话级、页面级、连接级、数据包级。

- **连接级**：
  - 网页包括多个目标，通过多个TCP连接进行传输
  - 参数例如：描述每个页面的连接个数，两个连续连接之间的时间，连接大小的分布
- **数据包级**：
  - 在TCP/IP数据包中对每个连接的字节进行划分
  - 参数例如：描述数据包大小的分布，数据包到达的时间间隔。

HTTP业务

1.1.1、HTTP业务模型

1.1.2、HTTP参数

1.1.3、NS3中实现HTTP业务

HTTP参数：会话级

会话级重要参数包括：

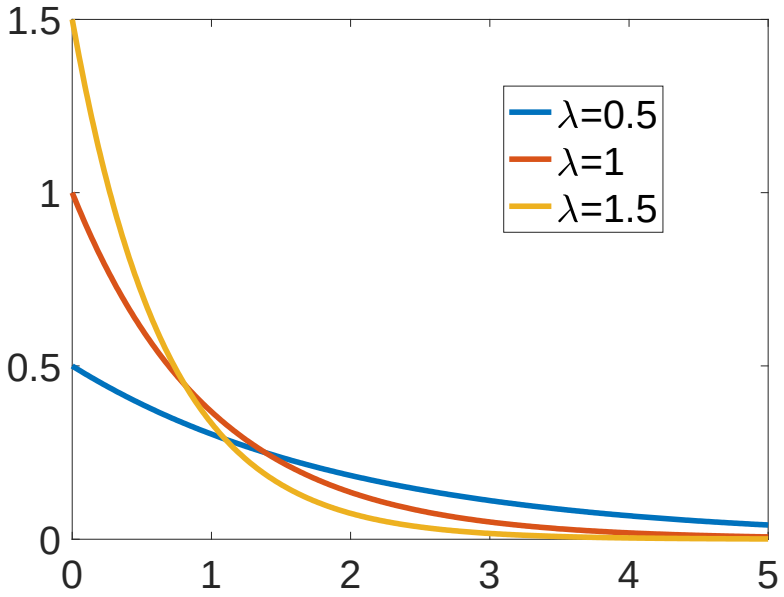
- 会话到达间隔
- 浏览时间
- 每个会话的页面数

- 会话到达间隔的常用分布：
  - 指数分布
  - 帕累托分布
- 浏览时间的常用分布：
  - Weibull分布
  - Gamma分布
  - 几何分布
- 每个会话页面数的常用分布：
  - Weibull分布
  - Lognormal分布
  - 几何分布等

HTTP参数：会话级

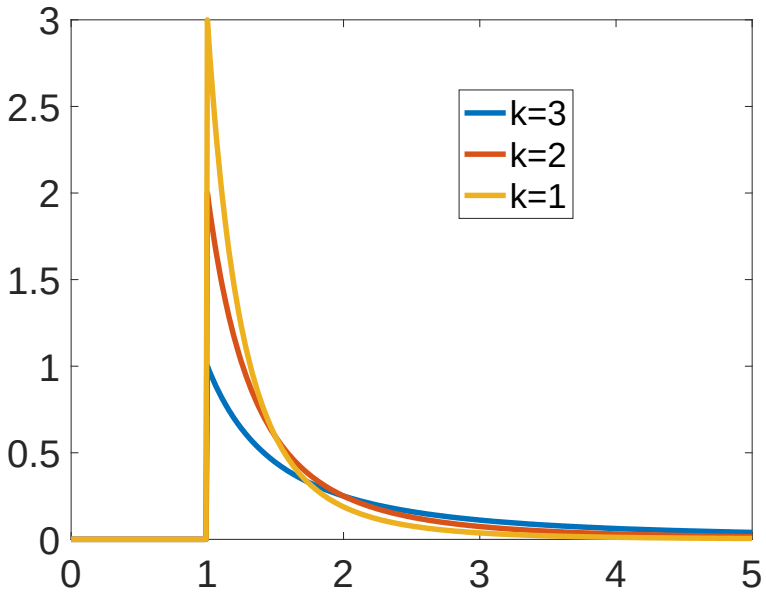
指数分布

$$f(x)=\begin{cases}\lambda e^{-\lambda x} & x>0\\ 0 & x\le 0\end{cases}$$



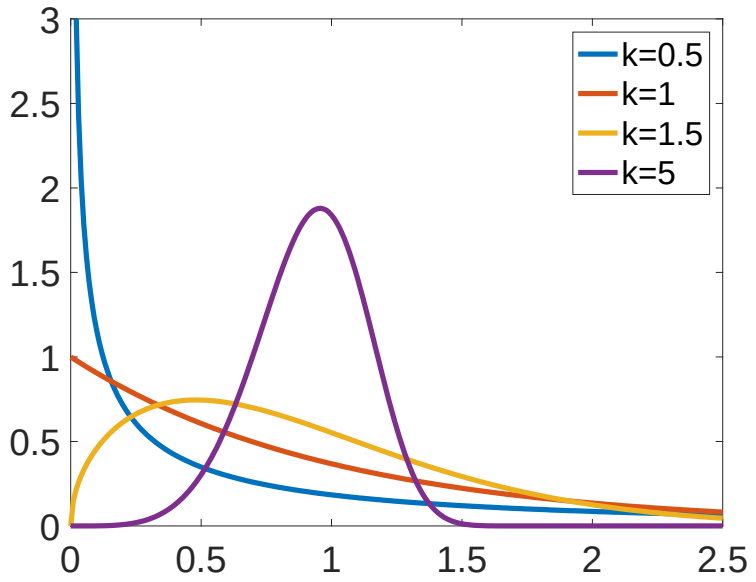
帕累托分布

$$p(x)=\begin{cases}0, & if\ x<x_{min}+\mu;\\ \dfrac{kx_{min}^{k}}{(x-\mu)^{k+1}}, & if\ x\ge x_{min}+\mu\end{cases}$$



Weibull分布

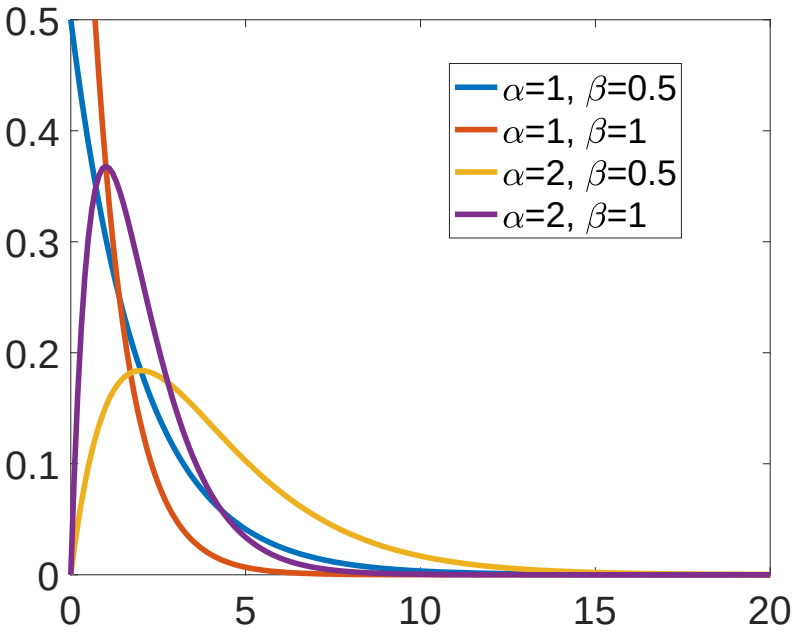
$$f(x;\lambda,k)=\begin{cases}\dfrac{k}{\lambda}\left(\dfrac{x}{\lambda}\right)^{k-1}e^{-(x/\lambda)^{k}} & x\ge 0\\ 0 & x<0\end{cases}$$



HTTP参数：会话级

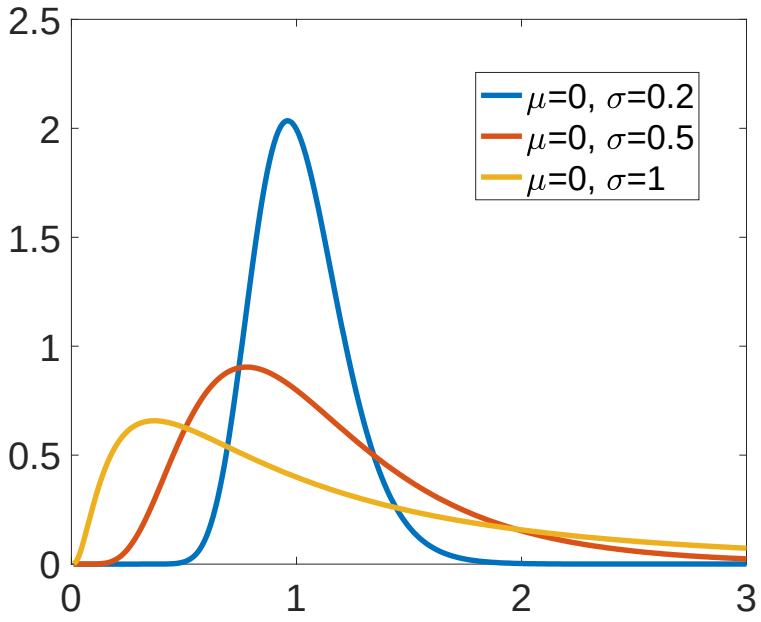
Gamma分布

$$f(x,\beta,\alpha)=\frac{\beta^{\alpha}}{\Gamma(\alpha)}x^{\alpha-1}e^{-\beta x},x>0$$



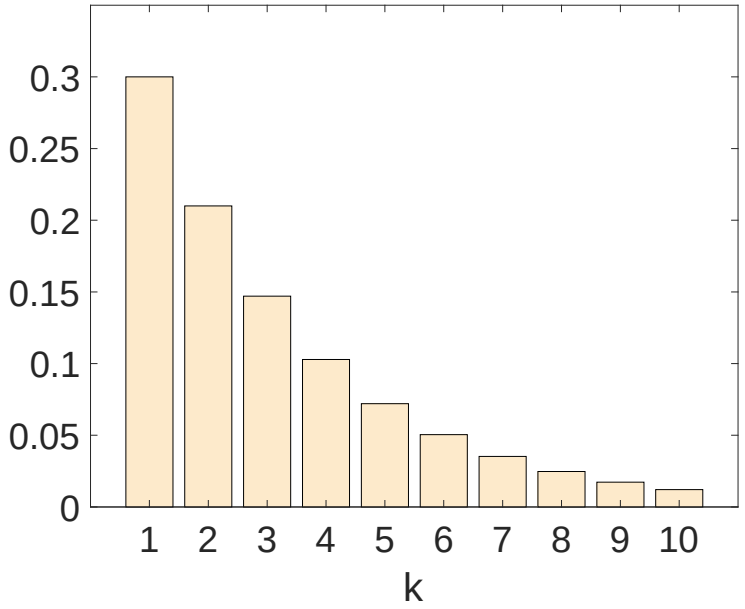
对数正态分布

$$p(x)=\frac{1}{x\sigma\sqrt{2\pi}}\mathrm{e}^{-\frac{(\ln x-u)^2}{2\sigma^2}}$$



几何分布

$$P(\mathrm{X}=k)=(1-p)^{\mathrm{k}-1}\mathrm{p},k=1,2,\ldots$$



HTTP参数：页面级

页面级重要**参数包括**：

- 同一会话中两个连续页面间隔时间
- 主对象特征
- 嵌入对象特征
- 主对象解析时间

- 同一会话中两个连续页面间隔时间的常用分布：
  - Weibull分布
  - Gamma分布
  - 帕累托分布
  - 几何分布
- 主对象大小的常用分布：
  - Lognormal分布
  - Pareto分布
- 嵌入对象个数的常用分布：
  - Gamma分布
  - Pareto分布
- 主对象解析时间的常用分布：
  - Weibull分布
  - 指数分布

HTTP参数：连接级

连接级重要参数包括：

- 每个页面的连接个数
- 同一页面两个连续连接的时间间隔
- 连接的大小

- 连接数的常用分布：
  - Gamma分布
  - 帕累托分布
- 时间间隔的常用分布：
  - Gamma分布
- 连接大小的常用分布：
  - Lognormal分布

HTTP业务

1.1.1、HTTP业务模型

1.1.2、HTTP参数

1.1.3、NS3中实现HTTP业务

01 业务产生器

包括了一到多个ThreeGppHttpClient应用，连接到一个ThreeGppHttpServer应用

02 客户端模型

对网页浏览器进行建模，从服务器请求网页

03 服务器模型

- 负责服务所请求的网页。
- 服务器基于所收到的请求类型，发送主对象（即网页的HTML文件）或者嵌入对象（例如HTML文件的图像）

3GPP HTTP客户端

- 该客户端是一种对网页服务器业务进行仿真的应用，该应用与**ThreeGppHttpServer**应用一起工作。
- 工作流程：

1. **打开**到目标网络服务器的**连接**
2. 连接建立后，立即发送请求数据包，从服务器**请求主对象**
3. 在收到**主对象**后，对其进行解析。
4. 解析过程确定网页中**嵌入式对象**的个数。
   - 如果有至少一个嵌入式对象，则从服务器请求第一个嵌入式对象。而后续嵌入式对象的请求会跟随前一个完全接收到的对象。
   - 如果没有嵌入式对象，则该应用进入阅读时间。
5. **阅读时间**需要一个较长的随机时延，此时不进行任何网络业务。
6. 阅读时间结束后，返回第2步。

3GPP HTTP服务器

- 对网页服务器业务进行仿真的应用，与**ThreeGppHttpClient**一起工作
- 响应请求：
  - 每个请求是一个数据包，包括了**ThreeGppHttpHeader**
  - 其中的属性content type决定了客户端所请求对象的类型，可能是主对象类型，或嵌入式对象类型
- 产生返回客户端的对象：
  - 所产生的对象大小是随机产生的
  - 每个对象可以以多个数据包的形式进行发送。
- **ThreeGppHttpServerTxBuffer**
  - 每个实体跟踪所服务的对象类型，以及剩下需要发送的字节数。
  - 该应用接收来自客户端的连接请求，每个连接一直处于保持状态直到客户端断开连接
  - **ThreeGppHttpVariables**配置最大传输单元大小。默认为536字节，最大为1460字节

3GPP HTTP——代码实现

3GPP HTTP客户端——分析主对象时间

分析主对象时间服从指数分布：

```
const Time parsingTime = m_httpVariables->GetParsingTime ();
```

```
ThreeGppHttpVariables::GetParsingTime ()
{
  return Seconds (m_parsingTimeRng->GetValue ());
}
```

```
m_parsingTimeRng = CreateObject<ExponentialRandomVariable> ();
```

指数分布的均值设置：

```
void
ThreeGppHttpVariables::SetParsingTimeMean (Time mean)
{
  NS_LOG_FUNCTION (this << mean.GetSeconds ());
  m_parsingTimeRng->SetAttribute ("Mean", DoubleValue (mean.GetSeconds ()));
}
```

3GPP HTTP客户端——主对象中嵌入对象数量

主对象中嵌入对象数量服从Pareto分布：

```
m_embeddedObjectsToBeRequested = m_httpVariables->GetNumOfEmbeddedObjects ();
ThreeGppHttpVariables::GetNumOfEmbeddedObjects ()
{// 参数验证
  const uint32_t upperBound =
    static_cast<uint32_t> (m_numOfEmbeddedObjectsRng->GetBound ());
  if (upperBound <= m_numOfEmbeddedObjectsScale){
      NS_FATAL_ERROR ("`NumOfEmbeddedObjectsMax` attribute "
                      << " must be greater than"
                      << " the `NumOfEmbeddedObjectsScale` attribute.");}
  //找一个随机值使它落入[scale, upperBound)区间，上面验证保证不会无限循环。
  uint32_t value;
  do{
      value = m_numOfEmbeddedObjectsRng->GetInteger ();
    }
  while ((value < m_numOfEmbeddedObjectsScale) || (value >= upperBound));
//返回值在区间[0, (upperBound - scale))内。
  return (value - m_numOfEmbeddedObjectsScale);}

m_numOfEmbeddedObjectsRng = CreateObject<ParetoRandomVariable> ();
```

3GPP HTTP客户端——主对象中嵌入对象数量

Pareto分布的max(μ)、shape($1/k$)、scale(x_{min})设置：

$$p(x) = \begin{cases} 0, & if\ x < x_{min} + \mu; \\ \dfrac{k x_{min}^{k}}{(x-\mu)^{k+1}}, & if\ x \geq x_{min} + \mu \end{cases}$$

```
void
ThreeGppHttpVariables::SetNumOfEmbeddedObjectsMax (uint32_t max)
{ NS_LOG_FUNCTION (this << max);
  m_numOfEmbeddedObjectsRng->SetAttribute ("Bound",
                                           DoubleValue (static_cast<double> (max)));}
void
ThreeGppHttpVariables::SetNumOfEmbeddedObjectsShape (double shape)
{ NS_LOG_FUNCTION (this << shape);
  NS_ASSERT_MSG (std::fabs (shape - 1.0) > 0.000001,
                 "Shape parameter must not equal to 1.0.");
  m_numOfEmbeddedObjectsRng->SetAttribute ("Shape", DoubleValue (shape));}
void
ThreeGppHttpVariables::SetNumOfEmbeddedObjectsScale (uint32_t scale)
{NS_LOG_FUNCTION (this << scale);
  NS_ASSERT_MSG (scale > 0, "Scale parameter must be greater than zero.");
  m_numOfEmbeddedObjectsScale = scale;
  m_numOfEmbeddedObjectsRng->SetAttribute ("Scale", DoubleValue (scale));}
```

3GPP HTTP客户端——阅读时间

阅读时间服从指数分布：

```
const Time readingTime = m_httpVariables->GetReadingTime ();
ThreeGppHttpVariables::GetReadingTime ()
{
  return Seconds (m_readingTimeRng->GetValue ());
}
m_readingTimeRng = CreateObject<ExponentialRandomVariable> ();
```

指数分布的均值设置：

```
void
ThreeGppHttpVariables::SetReadingTimeMean (Time mean)
{
  NS_LOG_FUNCTION (this << mean.GetSeconds ());
  m_readingTimeRng->SetAttribute ("Mean", DoubleValue (mean.GetSeconds ()));
}
```

3GPP HTTP服务端——MTU大小

MTU大小服从取值为1500和576的离散分布：

```
m_mtuSize = m_httpVariables->GetMtuSize ();
```

```
ThreeGppHttpVariables::GetMtuSize ()
{const double r = m_mtuSizeRng->GetValue ();
  NS_ASSERT (r >= 0.0);
  NS_ASSERT (r < 1.0);
  if (r < m_highMtuProbability) {
      return m_highMtu; // 1500 bytes if including TCP header.}
  else {
      return m_lowMtu; // 576 bytes if including TCP header. }}
```

```
m_mtuSizeRng = CreateObject<UniformRandomVariable> ();
```

3GPP HTTP服务端——MTU大小

MTU大小涉及到的代码参数（最大MTU、最小MTU、选择大MTU的概率）设置：

```
.AddAttribute ("LowMtuSize",
               "The lower MTU size.",
               UintegerValue (536),
               MakeUintegerAccessor (&ThreeGppHttpVariables::m_lowMtu),
               MakeUintegerChecker<uint32_t> (0))
.AddAttribute ("HighMtuSize",
               "The higher MTU size.",
               UintegerValue (1460),
               MakeUintegerAccessor (&ThreeGppHttpVariables::m_highMtu),
               MakeUintegerChecker<uint32_t> (0))
.AddAttribute ("HighMtuProbability",
               "The probability that higher MTU size is used.",
               DoubleValue (0.76),
               MakeDoubleAccessor (&ThreeGppHttpVariables::m_highMtuProbability),
               MakeDoubleChecker<double> (0, 1))
```

3GPP HTTP服务端——主对象大小

主对象大小服从对数正态分布：

```
const uint32_t objectSize = m_httpVariables->GetMainObjectSize ();
```

```
ThreeGppHttpVariables::GetMainObjectSize ()
{// 参数验证.
  if (m_mainObjectSizeMax <= m_mainObjectSizeMin)
    {NS_FATAL_ERROR ("`MainObjectSizeMax` attribute "
                     << " must be greater than"
                     << " the `MainObjectSizeMin` attribute.");  }
  //寻找一个值使它落在[min, max)区间内
  uint32_t value;
  do
    { value = m_mainObjectSizeRng->GetInteger (); }
  while ((value < m_mainObjectSizeMin) || (value >= m_mainObjectSizeMax));
  return value;}
```

```
m_mainObjectSizeRng = CreateObject<LogNormalRandomVariable> ();
```

3GPP HTTP服务端——主对象大小

对数正态分布参数（均值、方差）与代码中涉及的最大最小值的设置：

```
.AddAttribute ("MainObjectSizeMean",
               "The mean of main object sizes (in bytes).",
               UintegerValue (10710),
               MakeUintegerAccessor (&ThreeGppHttpVariables::SetMainObjectSizeMean),
               MakeUintegerChecker<uint32_t> ())
.AddAttribute ("MainObjectSizeStdDev",
               "The standard deviation of main object sizes (in bytes).",
               UintegerValue (25032),
               MakeUintegerAccessor (&ThreeGppHttpVariables::SetMainObjectSizeStdDev),
               MakeUintegerChecker<uint32_t> ())
.AddAttribute ("MainObjectSizeMin",
               "The minimum value of main object sizes (in bytes).",
               UintegerValue (100),
               MakeUintegerAccessor (&ThreeGppHttpVariables::m_mainObjectSizeMin),
               MakeUintegerChecker<uint32_t> (22))
.AddAttribute ("MainObjectSizeMax",
               "The maximum value of main object sizes (in bytes).",
               UintegerValue (2000000), // 2 MB
               MakeUintegerAccessor (&ThreeGppHttpVariables::m_mainObjectSizeMax),
               MakeUintegerChecker<uint32_t> ())
```

3GPP HTTP服务端——主对象大小

对数正态分布参数（均值、方差）设置：

```
void
ThreeGppHttpVariables::SetMainObjectSizeMean (uint32_t mean)
{ NS_LOG_FUNCTION (this << mean);
  NS_ASSERT_MSG (mean > 0, "Mean must be greater than zero.");
  m_mainObjectSizeMean = mean;
  if (IsInitialized ())
    { UpdateMainObjectMuAndSigma ();}}
void
ThreeGppHttpVariables::SetMainObjectSizeStdDev (uint32_t stdDev)
{ NS_LOG_FUNCTION (this << stdDev);
  m_mainObjectSizeStdDev = stdDev;
  if (IsInitialized ())
    {UpdateMainObjectMuAndSigma ();}}
```

3GPP HTTP服务端——主对象大小

对数正态分布参数更新：

```
void
ThreeGppHttpVariables::UpdateMainObjectMuAndSigma (void)
{
  NS_LOG_FUNCTION (this);
  const double a1 = std::pow (m_mainObjectSizeStdDev, 2.0);
  const double a2 = std::pow (m_mainObjectSizeMean, 2.0);
  const double a = std::log (1.0 + (a1 / a2));
  const double mu = std::log (m_mainObjectSizeMean) - (0.5 * a);
  const double sigma = std::sqrt (a);
  NS_LOG_DEBUG (this << " Mu= " << mu << " Sigma= " << sigma << ".");
  m_mainObjectSizeRng->SetAttribute ("Mu", DoubleValue (mu));
  m_mainObjectSizeRng->SetAttribute ("Sigma", DoubleValue (sigma));
}
```

3GPP HTTP服务端——嵌入对象大小

嵌入对象大小服从对数正态分布：

```
const uint32_t objectSize = m_httpVariables->GetEmbeddedObjectSize ();
```

```
ThreeGppHttpVariables::GetEmbeddedObjectSize ()
{ // 参数验证.
  if (m_embeddedObjectSizeMax <= m_embeddedObjectSizeMin)
    {NS_FATAL_ERROR ("`EmbeddedObjectSizeMax` attribute "
                     << " must be greater than"
                     << " the `EmbeddedObjectSizeMin` attribute.");  }
  //寻找一个值使它落在[min, max)区间内
  uint32_t value;
  do
  {value = m_embeddedObjectSizeRng->GetInteger ();}
  while ((value < m_embeddedObjectSizeMin) || (value >= m_embeddedObjectSizeMax))
  return value;}
```

```
m_embeddedObjectSizeRng = CreateObject<LogNormalRandomVariable> ();
```

3GPP HTTP服务端——嵌入对象大小

对数正态分布参数（均值、方差）与代码中涉及的最大最小值的设置：

```
.AddAttribute ("EmbeddedObjectSizeMean",
               "The mean of embedded object sizes (in bytes).",
               UintegerValue (7758),
               MakeUintegerAccessor (&ThreeGppHttpVariables::SetEmbeddedObjectSizeMean),
               MakeUintegerChecker<uint32_t> ())
.AddAttribute ("EmbeddedObjectSizeStdDev",
               "The standard deviation of embedded object sizes (in bytes).",
               UintegerValue (126168),
               MakeUintegerAccessor (&ThreeGppHttpVariables::SetEmbeddedObjectSizeStdDev),
               MakeUintegerChecker<uint32_t> ())
.AddAttribute ("EmbeddedObjectSizeMin",
               "The minimum value of embedded object sizes (in bytes).",
               UintegerValue (50),
               MakeUintegerAccessor (&ThreeGppHttpVariables::m_embeddedObjectSizeMin),
               MakeUintegerChecker<uint32_t> (22))
.AddAttribute ("EmbeddedObjectSizeMax",
               "The maximum value of embedded object sizes (in bytes).",
               UintegerValue (2000000), // 2 MB
               MakeUintegerAccessor (&ThreeGppHttpVariables::m_embeddedObjectSizeMax),
               MakeUintegerChecker<uint32_t> ())
```

3GPP HTTP服务端——嵌入对象大小

对数正态分布参数（均值、方差）设置：

```
void
ThreeGppHttpVariables::SetEmbeddedObjectSizeMean (uint32_t mean)
{NS_LOG_FUNCTION (this << mean);
  NS_ASSERT_MSG (mean > 0, "Mean must be greater than zero.");
  m_embeddedObjectSizeMean = mean;
  if (IsInitialized ())
    {  UpdateEmbeddedObjectMuAndSigma ();}}
void
ThreeGppHttpVariables::SetEmbeddedObjectSizeStdDev (uint32_t stdDev)
{NS_LOG_FUNCTION (this << stdDev);
  m_embeddedObjectSizeStdDev = stdDev;
  if (IsInitialized ())
    {UpdateEmbeddedObjectMuAndSigma (); }}
```

3GPP HTTP服务端——嵌入对象大小

对数正态分布参数（均值、方差）更新：

```
void
ThreeGppHttpVariables::UpdateEmbeddedObjectMuAndSigma (void)
{
  NS_LOG_FUNCTION (this);
  const double a1 = std::pow (m_embeddedObjectSizeStdDev, 2.0);
  const double a2 = std::pow (m_embeddedObjectSizeMean, 2.0);
  const double a = std::log (1.0 + (a1 / a2));
  const double mu = std::log (m_embeddedObjectSizeMean) - (0.5 * a);
  const double sigma = std::sqrt (a);
  NS_LOG_DEBUG (this << " Mu= " << mu << " Sigma= " << sigma << ".");
  m_embeddedObjectSizeRng->SetAttribute ("Mu", DoubleValue (mu));
  m_embeddedObjectSizeRng->SetAttribute ("Sigma", DoubleValue (sigma));
}
```

模型用法

- **ThreeGppHttpServerHelper**
- **ThreeGppHttpClientHelper**
- **ThreeGppHttpServer**
- **ThreeGppHttpClient**
- **ThreeGppHttpVariables**

- 使用ThreeGppHttpServerHelper和ThreeGppHttpClientHelper，实现在节点上安装ThreeGppHttpServer和ThreeGppHttpClient应用。
- Helper对象可以对客户端和服务器对象进行属性配置，但无法对ThreeGppHttpVariables对象进行配置。
- ThreeGppHttpVariables的配置通过直接修改其属性来实现，必须在安装应用到节点之前进行
- HTTP应用的运行例子：$ ./waf --run 'three-gpp-http-example'，该例子输出客户端的网页请求和服务器端的响应。

目录

拓扑建模

01、拓扑建模基本理论

02、NS3中实现拓扑建模

拓扑建模的基本理论

拓扑建模的必要性

- 网络拓扑是信息网络的一个主要组成部分。
- 描述了网络实体如何彼此直接互连，以及信息的流动方式。

固定拓扑与随机拓扑

- 很多网络拓扑结构是固定不变的。
  - 如点对点、总线、星形等；
- 有的网络拓扑与节点的地理位置密切相关
  - 移动通信网
  - 自组织网络
- 网络中节点的生灭会造成网络拓扑结构的变化

固定拓扑模型

- 直接建立节点之间的连接关系
- 节点之间的传输通道质量由速率、时延和信道模型等其他参数给出

随机拓扑模型

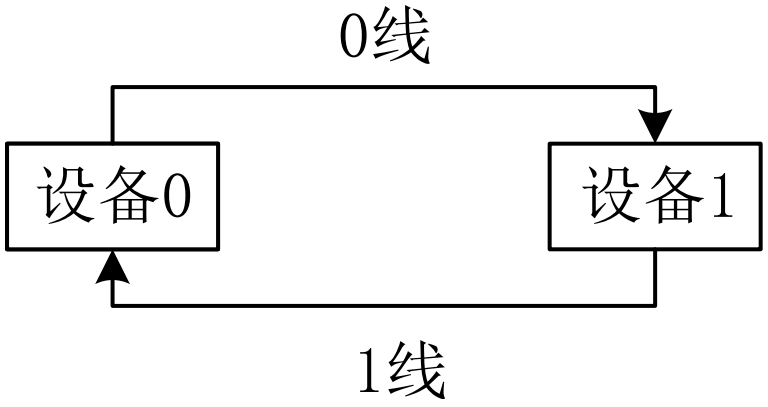
- 通常涉及到区域内撒点问题；
- 先根据随机分布确定节点位置，再根据位置确定拓扑连接关系。

拓扑建模

01、拓扑建模基本理论

02、NS3中实现拓扑建模

P2P拓扑模型



- 点对点信道连接图：点对点信道将两个网络设备直接相连，并为设备分配0线和1线。

- **P2P模型用于两个节点之间的点到点分组传输**
  - 信道只被发送端和接收端两个节点共享，因此只要当前没有正在传输的数据，发送端就可以持续地向信道发送分组。

P2P拓扑模型

- 在NS3中，P2P模型主要是通过PointToPointHelper助手类来完成。
- 该例子位于src/point-to-point/model文件夹下，可直接调用。
- 一部分代码：

```
PointToPointChannel::Attach (Ptr<PointToPointNetDevice> device)
{
  NS_LOG_FUNCTION (this << device);
  //最多连接两个点对点网络设备
  NS_ASSERT_MSG (m_nDevices < N_DEVICES, "Only two devices
permitted");
  NS_ASSERT (device != 0);
  //为设备分配0线和1线
  m_link[m_nDevices++].m_src = device;
  if (m_nDevices == N_DEVICES)
    {  m_link[0].m_dst = m_link[1].m_src;  //0线的传输终点设为设备1
       m_link[1].m_dst = m_link[0].m_src;  //1线的传输终点设为设备0
       m_link[0].m_state = IDLE;
```

总线型拓扑模型

拓扑结构定义

采用单根数据传输线作为通信介质，所有的节点都通过相应的硬件接口直接连接到通信介质，而且能被所有其他的节点接收

拓扑结构组成

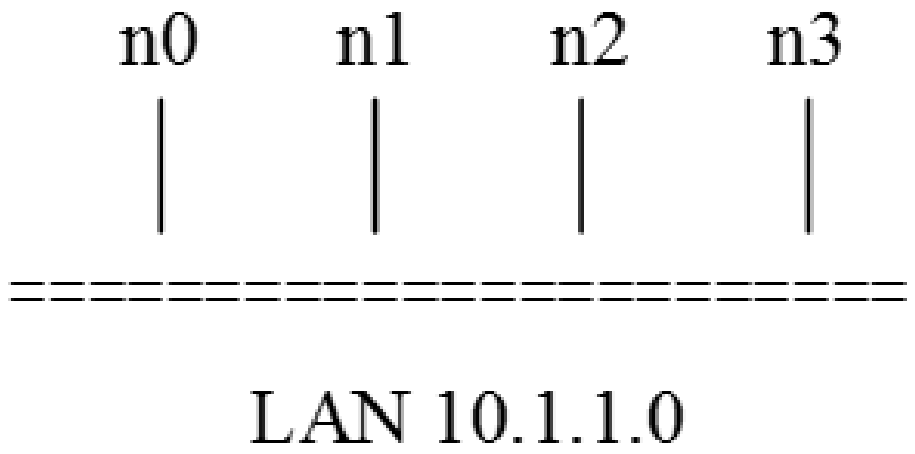
- 用户节点：服务器或工作站
- 通信介质：铜线、同轴电缆、光纤

所有的节点共享一条公用的传输链路，所以一次只能由一个设备传输

拓扑结构控制策略

总线型网络采用载波监听多路访问/冲突检测协议（CSMA/CD）作为控制策略

总线型拓扑模型



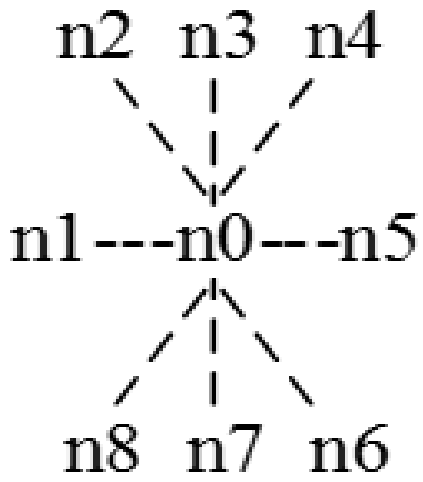
4节点CSMA网络的拓扑结构

- 在NS3中，CSMA网络设备模型包含5个类：
  - Backoff类
  - CsmaChannel类
  - CsmaDeviceRec类
  - CsmaHelper类
  - CsmaNetDevice类

星形网络拓扑模型

- 星形网络：各工作站以星型方式连接成网。
- 网络有中心节点，其他节点（工作站、服务器）都与中心节点直接相连，由中心节点向目的节点发送数据，任意两个节点间的通信都需要通过中心节点。
- 星型结构是一种广泛使用的拓扑结构。
  - 如：一个AP及其控制下的所有WiFi终端

星型网络拓扑模型



星型网络拓扑结构示意图

- 建立步骤和CSMA拓扑类似：
  - 创建节点、网络协议栈、信道、网络设备
  - 配置IP地址
- 主要的区别：
  - 星型拓扑结构在创建节点时，需要把中心节点和周围节点分开创建，以达到将服务器和客户端进行区分的目的
  - 假如周围节点作为服务器接收数据包，则中心节点作为客户端时，对应不同服务器将有不同的IP地址，在配置IP地址时需要分别设置

例子：　拓扑建模：examples/tutorial/third.cc

```
// 创建一个移动助手类，并通过“位置分配器”分配最初的移动节点位置。
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                    “MinX”, DoubleValue (0.0),  //起点x坐标
                    “MinY”, DoubleValue (0.0), //起点y坐标
                    “DeltaX”, DoubleValue (5.0), //x增量
                    “DeltaY”, DoubleValue (10.0), //y增量
                    "GridWidth", UintegerValue (3), //每一行每一列上的网格数量
                    “LayoutType”, StringValue (“RowFirst”)//布局类型，先分配行还是先分配列
                        ); //设置坐标系和移动节点位置
```

例子： 节点的位置分配

调用”SetPositionAllocator”生成一个位置分配器对象。

```
52  void
53  MobilityHelper::SetPositionAllocator (std::string type,//位置分配器的类型
54                                        std::string n1, const AttributeValue &v1,//参数1的名字和数值
55                                        std::string n2, const AttributeValue &v2,
56                                        std::string n3, const AttributeValue &v3,
57                                        std::string n4, const AttributeValue &v4,
58                                        std::string n5, const AttributeValue &v5,
59                                        std::string n6, const AttributeValue &v6,
60                                        std::string n7, const AttributeValue &v7,
61                                        std::string n8, const AttributeValue &v8,
62                                        std::string n9, const AttributeValue &v9)
63  {
64    ObjectFactory pos;
65    pos.SetTypeId (type);//设置对象的类别
66    pos.Set (n1, v1);//设置参数的数值
67    pos.Set (n2, v2);
68    pos.Set (n3, v3);
69    pos.Set (n4, v4);
70    pos.Set (n5, v5);
71    pos.Set (n6, v6);
72    pos.Set (n7, v7);
73    pos.Set (n8, v8);
74    pos.Set (n9, v9);
75    m_position = pos.Create ()->GetObject<PositionAllocator> ();//创建对应的位置分配器对象
76  }
```

NS-3中的运动模型

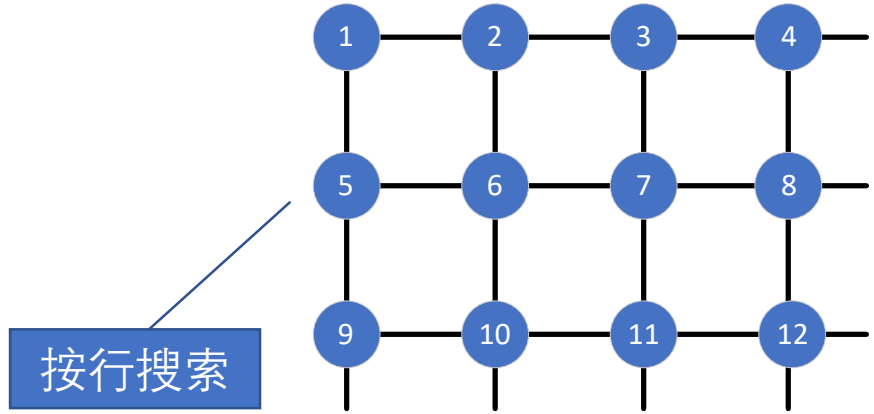
例子：　节点的位置分配　　调用GridPositionAllocator中的GetNext方法获取节点的具体位置

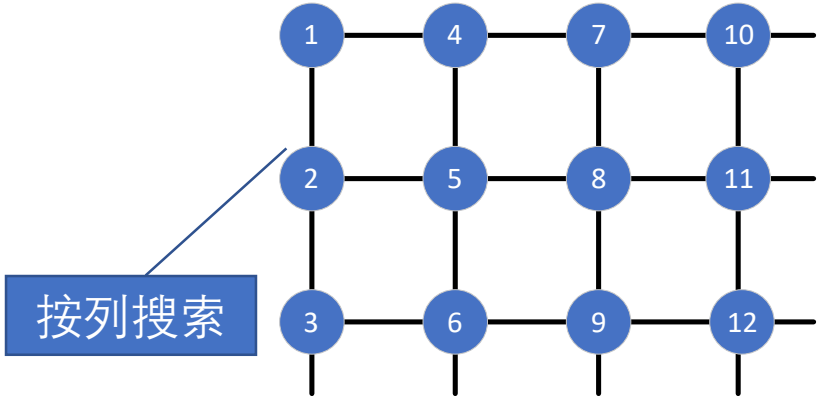
```
130 MobilityHelper::Install (Ptr<Node> node) const
131 {
132   Ptr<Object> object = node;
133   Ptr<MobilityModel> model = object->GetObject<MobilityModel> ();
134   if (model == 0)
135     {
136       model = m_mobility.Create ()->GetObject<MobilityModel> ();
137       if (model == 0)
138         {
139           NS_FATAL_ERROR ("The requested mobility model is not a mobility model: \""<<
140                           m_mobility.GetTypeId ().GetName ()<<"\"");
141         }
142       if (m_mobilityStack.empty ())
143         {
144           NS_LOG_DEBUG ("node="<<object<<", mob="<<model);
145           object->AggregateObject (model);
146         }
147       else
148         {
149           // we need to setup a hierarchical mobility model
150           Ptr<MobilityModel> parent = m_mobilityStack.back ();
151           Ptr<MobilityModel> hierarchical =
152             CreateObjectWithAttributes<HierarchicalMobilityModel> ("Child", PointerValue (model),
153                                                                    "Parent", PointerValue (parent));
154           object->AggregateObject (hierarchical);
155           NS_LOG_DEBUG ("node="<<object<<", mob="<<hierarchical);
156         }
157     }
158   Vector position = m_position->GetNext ();//获取网格下一位置
159   model->SetPosition (position);//将位置赋予节点
160 }
```

例子：　• 节点的位置分配

GridPositionAllocator中的GetNext方法的具体实现

```
210  Vector
211  GridPositionAllocator::GetNext (void) const
212  {
213    double x = 0.0, y = 0.0;
214    switch (m_layoutType) {
215      case ROW_FIRST://沿着行寻找下一个位置
216        x = m_xMin + m_deltaX * (m_current % m_n);//计算当前位置的行坐标
217        y = m_yMin + m_deltaY * (m_current / m_n);//计算当前位置的列坐标
218        break;
219      case COLUMN_FIRST://沿着列寻找
220        x = m_xMin + m_deltaX * (m_current / m_n);
221        y = m_yMin + m_deltaY * (m_current % m_n);
222        break;
223      }
224    m_current++;
225    return Vector (x, y, m_z);
226  }
```





目录

运动模型建模

3.1、运动建模基本理论

3.2、在NS-3中实现运动建模

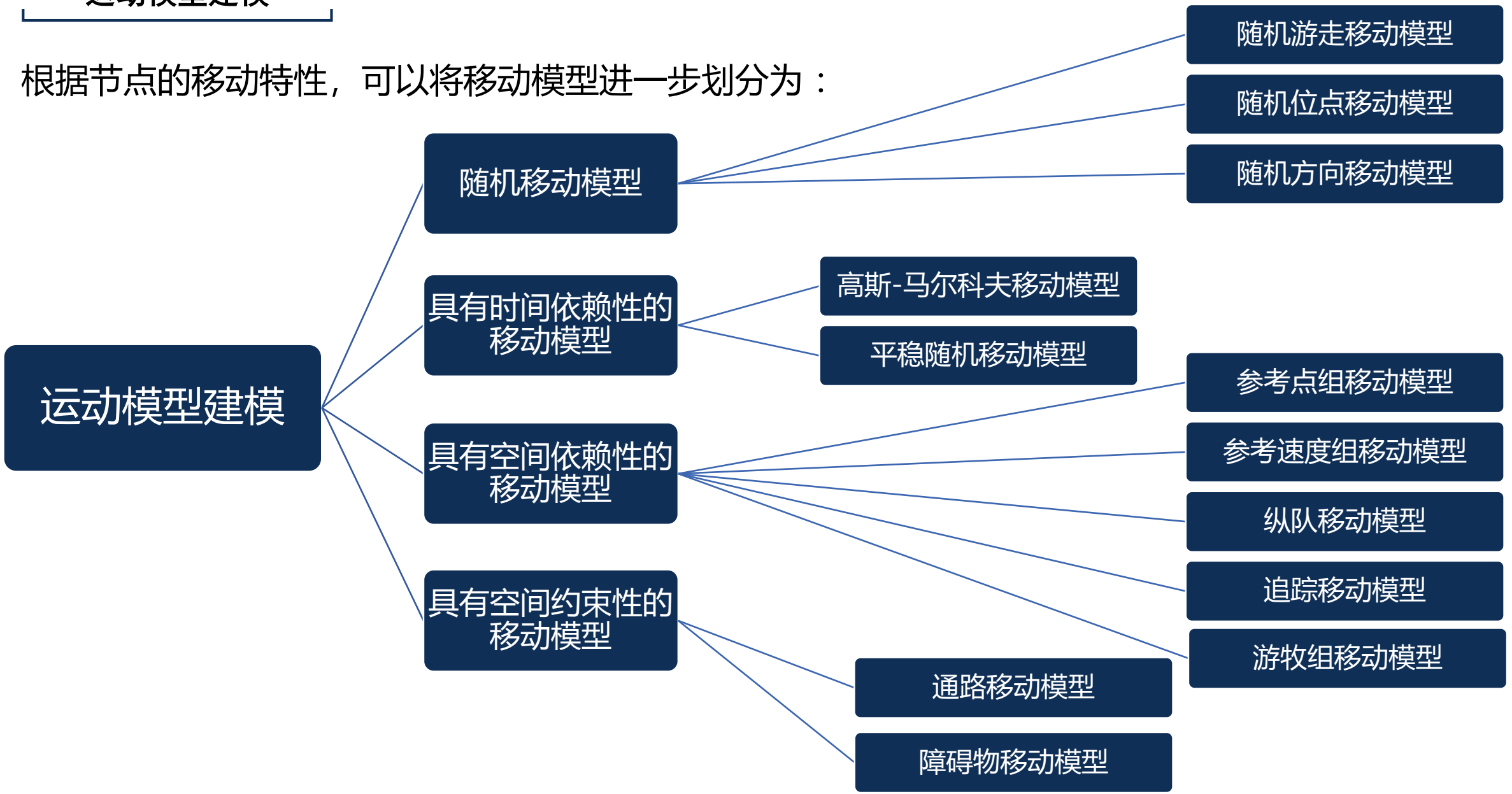
运动模型建模的必要性

- 在无线网络中，通信可以基于基础设施（例如，WLAN接入点或基站）进行，或者可以在Ad hoc模式下进行，其中的移动设备彼此直接连接。
- 对这些无线网络进行研究并模拟时，通常**需要考虑模拟环境内实体的移动**，因此需要考虑对实体的运动模型进行建模。

运动模型的选择

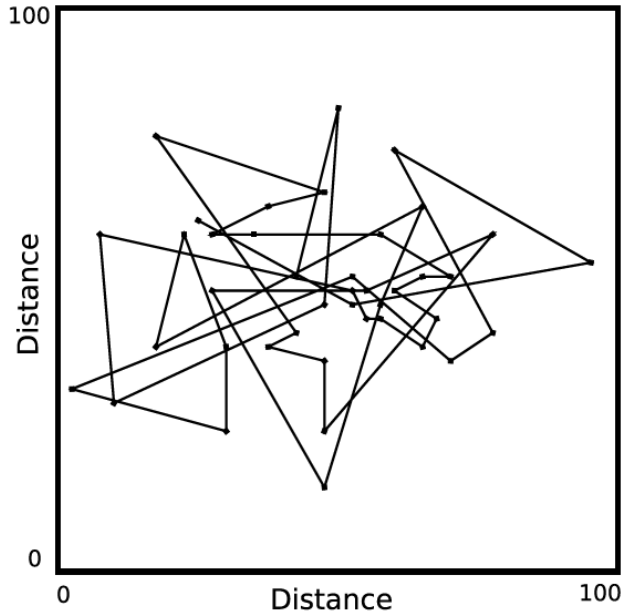
- 运动模型的选择必须在**准确性与其计算成本**之间进行权衡。
- 运动模型越**精确**，越适合建模现实场景，会产生更真实的结果。然而，这种现实的运动模型通常会带来**高复杂性**。

根据节点的移动特性，可以将移动模型进一步划分为：



（1）随机游走移动模型

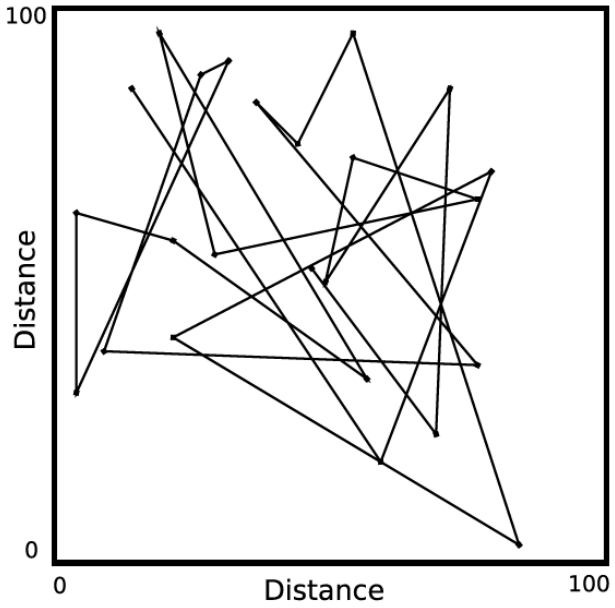
- 一种广泛使用的模型，根据自然界中的实体以不可预测的方式移动而建立的。
  - 每个实体**随机选择速度和方向移动**，直到走过固定的时间量（也可采用固定距离的方式）
  - 方向和速度限于预定范围，每个动作都限制在一个恒定的时间间隔内。
  - 移动后，对下一次移动的方向和速度进行计算。
  - 当实体击中模型的一个边界（由矩形指定），会以一定的反射角度和速度在边界上反弹
  - 一种无记忆移动模式，当前运动的速度和方向完全独立于其过去的速度和方向。
  - 该模型通常被称为布朗运动模型。



（2）随机位点移动模型

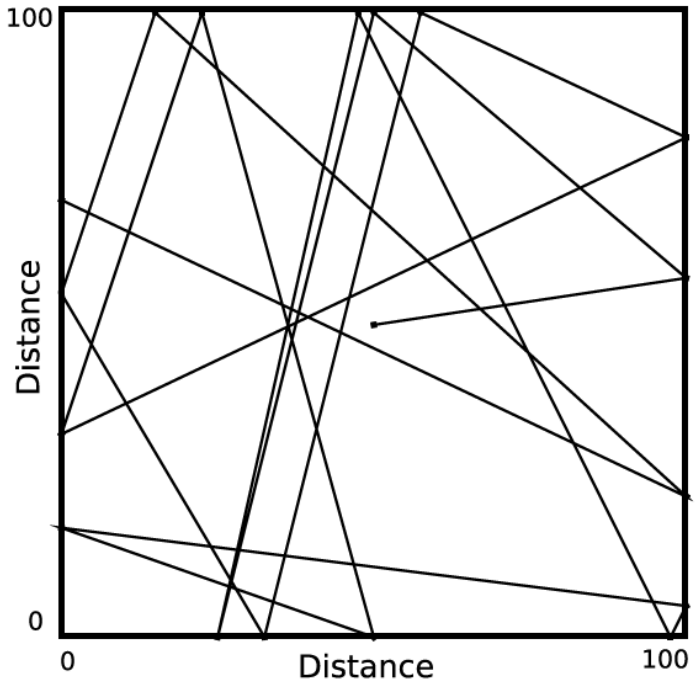
- 随机位点运动模型也是一种广泛使用的模型，与随机游走运动模型非常相似。

  - 实体选择随机目的地坐标（在模拟区域内）和随机速度（在限定范围内），然后从当前位置移动到目标位置。
  - 随机位点运动模型定义了两次运动之间的暂停时间。暂停后，模型将计算新的运动形式。
  - 随机位点是按照既定目标点的位置进行运动
  - 如果暂停时间设置为零并且速度范围选择相似，则随机位点运动模型与随机游走运动模型类似



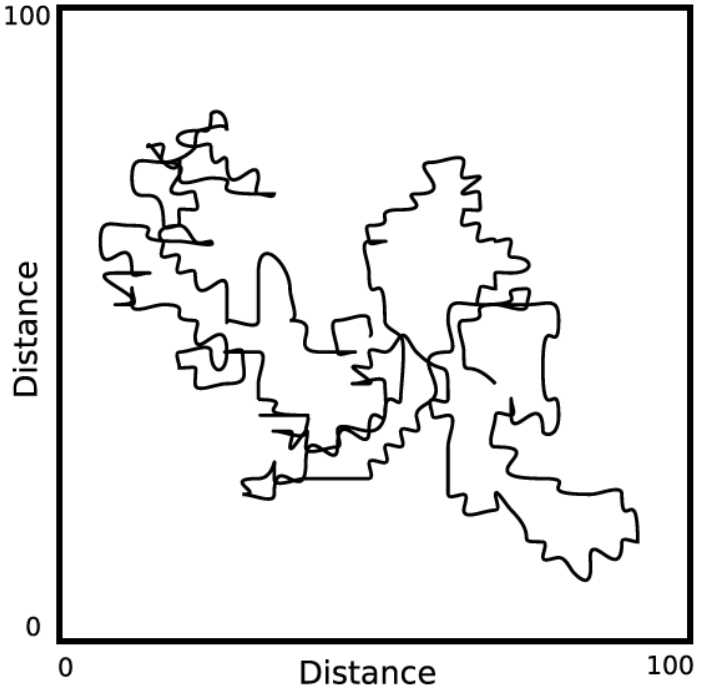
（3）随机方向移动模型

- 移动流程：
  - 实体随机选择一个方向和速度。
  - 该实体基于所选择的方向和速度移动，直至其到达仿真区域的边界。
  - 在到达边界后，该实体将停留一段预定的时间，然后选择新的方向和速度重新移动。



（4）高斯马尔科夫移动模型

- 高斯马尔科夫模型具有记忆性和可变性，模型的运动可以依赖于先前运动。
  - 运动实体被分配初始速度和方向，在固定的时间间隔内对实体进行方向和速度的更新。
  - 定义参数α (α=[0,1])来控制当前运动对先前运动的依赖程度：
    - ✓ 当α=0时，新运动不依赖于先前的运动和结果，类似于随机游走；
    - ✓ 当0<α<1时，获得中间水平的随机性；
    - ✓ 当α= 1时，实体以线性方式运动。

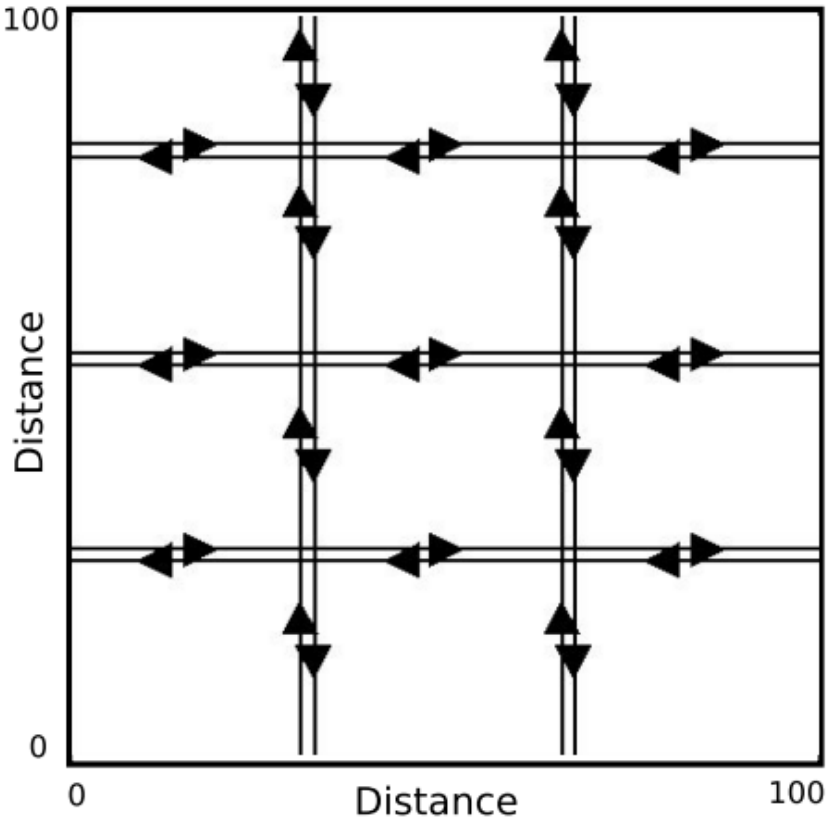


- 可以为实体指定运动的平均速度。
- 为了避免与模拟区域的边界发生碰撞，当实体接近边界时，可以调整实体方向：当实体到边界的距离小于一定值时，使实体离开边界。

(4) 曼哈顿模型

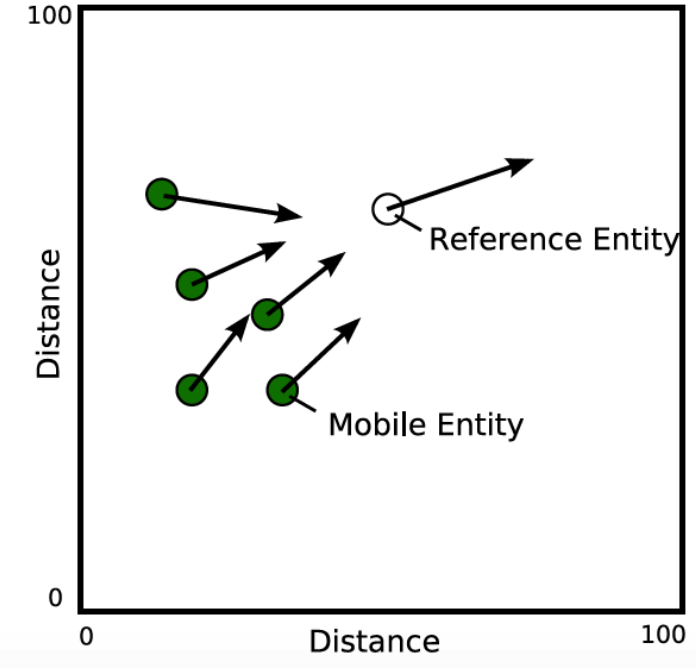
曼哈顿模型是一种广泛应用的街区或者高速公路移动模型

- 实体沿着双向的街道和十字路口移动。
- 需定义速度、加速度、安全距离
- 实体以一定的概率沿着街道或者十字路口移动
  - ✓ 以0.5的概率直行
  - ✓ 以0.25的概率左转弯
  - ✓ 以0.25的概率右转弯



（6）追踪移动模型（Pursue Mobility）

- 一种群体移动模型，其中一组移动实体正在追踪单个参考实体。
  - 参考实体使用的是实体移动模型，例如随机游走移动模型。
  - 其他实体都追踪参考实体，但是在它们的移动方向上增加了较小的偏差。
- 在该模型中模拟了加速度。
- 该模型的典型例子如：一群跟随博物馆向导进行移动的游客们。



运动模型建模

3.1、运动建模基本理论

3.2、NS-3中实现的随机游走模型

2D随机游走运动模型

随机游走运动模型对应的类为RandomWalk2dMobilityModel类，具有以下属性：

- Bounds：游走区域的界限。
- Time：在此延迟移动后更改当前方向和速度。
- Distance：移动此距离后，改变当前的方向和速度。
- Mode：指示用于改变当前速度和方向条件的模式。
- Direction：用于选择方向的随机变量（弧度）。
- Speed：用于选择速度（m/s）的随机变量。

NS-3中的运动模型

例子：

- 随机游走运动模型：examples/tutorial/third.cc
- 该模型中设置了运动的边界，其他属性皆为默认

//首先创建一个移动助手类，并通过“位置分配器”分配最初的移动节点位置。然后令节点以随机游走的方式移动。最后将移动模型安装到移动节点上。

```
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                    “MinX”, DoubleValue (0.0),  //起点x坐标
                    “MinY”, DoubleValue (0.0), //起点y坐标
                    “DeltaX”, DoubleValue (5.0), //x增量
                    “DeltaY”, DoubleValue (10.0), //y增量
                    "GridWidth", UintegerValue (3), //每一行每一列上的网格数量
                    “LayoutType”, StringValue (“RowFirst”)//布局类型，先分配行还是先分配列
                        ); //设置坐标系和移动节点位置
mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
                    "Bounds", RectangleValue (Rectangle (-50, 50, -50, 50)));
//使节点在指定大小的长方形区域以随机的速度和方向运动
mobility.Install (wifiStaNodes);
```

例子：　• 随机游走运动模型

SetMobilityModel内部代码

```
void
MobilityHelper::SetMobilityModel(std::string type, // 运动模型的类型
                                 std::string n1, const AttributeValue &v1, // 属性的名字和数值
                                 std::string n2, const AttributeValue &v2,
                                 std::string n3, const AttributeValue &v3,
                                 std::string n4, const AttributeValue &v4,
                                 std::string n5, const AttributeValue &v5,
                                 std::string n6, const AttributeValue &v6,
                                 std::string n7, const AttributeValue &v7,
                                 std::string n8, const AttributeValue &v8,
                                 std::string n9, const AttributeValue &v9)
 {
   m_mobility.SetTypeId(type); // 设置对象类别
   m_mobility.Set(n1, v1);   // 设置参数数值
   m_mobility.Set(n2, v2);
   m_mobility.Set(n3, v3);
   m_mobility.Set(n4, v4);
   m_mobility.Set(n5, v5);
   m_mobility.Set(n6, v6);
   m_mobility.Set(n7, v7);
   m_mobility.Set(n8, v8);
   m_mobility.Set(n9, v9);
 }
```

例子：　• 随机游走运动模型

调用” SetMobilityModel”生成一个运动对象

```
mobility.SetMobilityModel (“ns3::RandomWalk2dMobilityModel”, //调用RandomWalk2dMobilityModel.cc
                           “Mode”, StringValue (“Time”),
                           //改变当前速度和方向条件的模式是时间模式
                           “Time”, StringValue (“2s”),
                           //时延
                          "Speed", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"),
                           //速度
                           “Bounds”, StringValue (“0|200|0|200”)); //边界范围
```

例子：　• 随机游走运动模型

// RandomWalk2dMobilityModel.cc内部代码：随机运动模型的默认属性

```
.AddAttribute ("Bounds", // 属性名
               "Bounds of the area to cruise.",
               RectangleValue (Rectangle (0.0, 100.0, 0.0, 100.0)), // 模拟区域的边界范围
               MakeRectangleAccessor (&RandomWalk2dMobilityModel::m_bounds),
               MakeRectangleChecker ())
.AddAttribute ("Time",
               "Change current direction and speed after moving for this delay.",
               TimeValue (Seconds (1.0)), // 延迟1.0s后改变当前的方向和速度
               MakeTimeAccessor (&RandomWalk2dMobilityModel::m_modeTime),
               MakeTimeChecker ())
.AddAttribute ("Distance",
               "Change current direction and speed after moving for this distance.",
               DoubleValue (1.0), // 移动1.0米后改变运动方向和速度
               MakeDoubleAccessor (&RandomWalk2dMobilityModel::m_modeDistance),
               MakeDoubleChecker<double> ())
```

NS-3中的运动模型

例子：• 随机游走运动模型

// RandomWalk2dMobilityModel.cc内部代码：随机运动模型的默认属性

```
.AddAttribute ("Mode", //指示用于改变当前速度和方向条件的模式
               "The mode indicates the condition used to "
               "change the current speed and direction",
               EnumValue (RandomWalk2dMobilityModel::MODE_DISTANCE),
               MakeEnumAccessor (&RandomWalk2dMobilityModel::m_mode),
               MakeEnumChecker (RandomWalk2dMobilityModel::MODE_DISTANCE, “Distance”, // 距离模式
                                RandomWalk2dMobilityModel::MODE_TIME, “Time”)) // 时间模式
.AddAttribute ("Direction",
               "A random variable used to pick the direction (radians).",
               StringValue ("ns3::UniformRandomVariable[Min=0.0|Max=6.283184]"),
               // 方向(弧度)的范围在[0,2*pi]内随机产生
               MakePointerAccessor (&RandomWalk2dMobilityModel::m_direction),
               MakePointerChecker<RandomVariableStream> ())
.AddAttribute ("Speed",
               "A random variable used to pick the speed (m/s).",
               StringValue ("ns3::UniformRandomVariable[Min=2.0|Max=4.0]"),// 设定速度在[2,4]内随机产生

               MakePointerAccessor (&RandomWalk2dMobilityModel::m_speed),
               MakePointerChecker<RandomVariableStream> ());
```

例子：　• 随机游走运动模型：用代码描述随机游走的过程

```
//初始化
RandomWalk2dMobilityModel::DoInitializePrivate (void) {
m_helper.Update ();  // 更新位置
  double speed = m_speed->GetValue ();   // 给速度赋值
  double direction = m_direction->GetValue ();  // 给方向赋值
  Vector vector (std::cos (direction) * speed,
                 std::sin (direction) * speed,
                 0.0);
  m_helper.SetVelocity (vector);  // 设置新的速度向量
  m_helper.Unpause(); // 从当前位置以当前速度恢复移动
  Time delayLeft;
  if (m_mode == RandomWalk2dMobilityModel::MODE_TIME)  // 时间模式
    {
      delayLeft = m_modeTime;   // 时间模式下，时延为设定的时间
    }
  else
    {
      delayLeft = Seconds (m_modeDistance / speed);  // 距离模式下，时延为走过固定距离的时间
    }
  DoWalk (delayLeft); } // 开始游走
```

NS-3中的运动模型

例子：• 随机游走运动模型：

// 随机游走

```
 RandomWalk2dMobilityModel::DoWalk (Time delayLeft) { // 根据位置和速度行走，直到到达距离，到达时间，或与边界相交
Vector position = m_helper.GetCurrentPosition (); // 获取当前位置向量
  Vector speed = m_helper.GetVelocity ();  // 获取当前时间向量
  Vector nextPosition = position;
  nextPosition.x += speed.x * delayLeft.GetSeconds ();
  nextPosition.y += speed.y * delayLeft.GetSeconds ();  // x和y方向上的位置
  m_event.Cancel (); // 取消事件
  if (m_bounds.IsInside (nextPosition))
    {
      m_event = Simulator::Schedule (delayLeft, &RandomWalk2dMobilityModel::DoInitializePrivate, this);
    }
  else
    {
      nextPosition = m_bounds.CalculateIntersection (position, speed);  // 返回值为矩形与当前速度矢量的交点
//m_bounds为一个2D矩形，CalculateIntersection的参数为当前的位置和速度，返回值为矩形与当前速度矢量的交点
      Time delay = Seconds ((nextPosition.x - position.x) / speed.x); // 时延
      m_event = Simulator::Schedule (delay, &RandomWalk2dMobilityModel::Rebound, this, // 检查边界
                                     delayLeft - delay);
    }
  NotifyCourseChange (); // 路线改变
}
```

例子：　• 随机游走运动模型：

// 如果节点到达边界，则执行节点的反弹

```
RandomWalk2dMobilityModel::Rebound (Time delayLeft) // 如果节点到达边界，则执行节点的反弹
{
  m_helper.UpdateWithBounds (m_bounds); // 从上次更新的位置和时间更新位置(如果没有暂停)
  Vector position = m_helper.GetCurrentPosition (); // 获取当前位置
  Vector speed = m_helper.GetVelocity (); // 获取速度;如果暂停，将返回一个零向量
  switch (m_bounds.GetClosestSide (position)) // 参数：测试的位置；   返回：输入位置最接近的矩形的边
    {
    case Rectangle::RIGHT:
    case Rectangle::LEFT:
      speed.x = -speed.x;  // 到达左/右边，x方向速度方向转变
      break;
    case Rectangle::TOP:
    case Rectangle::BOTTOM:
      speed.y = -speed.y; // 到达上/下边，y方向速度方向转变
      break;
    }
  m_helper.SetVelocity (speed);  // 设置新的速度
  m_helper.Unpause ();  // 从当前位置以当前速度恢复移动
  DoWalk (delayLeft); //开始游走
}
```

目录

信道模型建模

5.1、信道建模基本理论

5.2、在NS-3中实现信道建模

- 信道建模必要性：
  - 通信节点之间的数据收发不是理想的。
  - 信息网络建模与仿真中必须包含对这种非理想性的模拟。

- 信道模型的分类：
  - 波形信道：用于链路级仿真，直接对电磁波传播所带来的多径传播等效应进行建模。
  - 抽象信道：不直接对信道中的物理过程进行建模波形，而是对信道所关注的层次上反映出来的一些指标进行抽象
    - 误码率
    - 误码块率
    - 可达性

波形信道建模实例：Jake模型

- 实现单路径固定Jakes传播损耗模型：
  - 考虑一个移动发射器 - 接收器对，假设接收机接收到M条不可分辨多径，每个多径都对应不同的到达方向，到达方向在[0,2π]区间上均匀分布，各径平均功率相等。

计算这种情况下的复系数如下：

$$X(t) = X_c(t) + jX_s(t)$$

- $X_c(t) = \frac{2}{\sqrt{M}} \sum_{n=1}^{M} \cos(\psi_n) \cos(\omega_d \cos(a_n) + \phi_n)$，
- $X_s(t) = \frac{2}{\sqrt{M}} \sum_{n=1}^{M} \sin(\psi_n) cos(\omega_d cos(a_n) + \phi_n)$

- 抽象信道模型中噪声的计算
  - 无论将信道抽象为误码率、误帧率、误块率还是可达性，其关键都是获得**等效SNR**；
  - 在一般的链路中，噪声只考虑热噪声，可以利用热噪声功率谱密度和带宽计算得到。
  - 热噪声功率谱密度又取决于绝对温度T。

$$N_0 = kT$$

  - 玻尔兹曼常数: k= 1.38×10^{-23} J/K
  - 温度取常温17摄氏度: T=290K
  - 功率谱密度: $N_0 = kT \approx 4.00 \times 10^{-21}$J=$4.00 \times 10^{-21}$W/Hz
  - 通常用dB值表示: $N_0 =$ $-204\ dB$W/Hz = $-174\ dBm$/Hz

- 抽象信道模型中噪声的计算
  - 在已知信号带宽B Hz时，可将带宽也转换为dB值，计算出热噪声功率

$$P_n = -174\ dBm/\text{Hz}+10\log10(B)$$

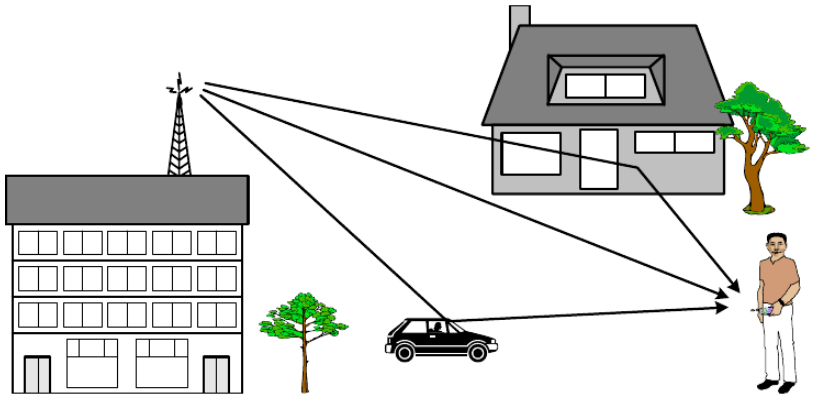
  - 一般考虑通信接收机具有噪声系数，相当于在上面计算的噪声功率以倍数形式增大了噪声，也用dB值表示，噪声系数通常取4~10dB
  - 这样，我们就可以得到总的接收机噪声功率，只要得到信号功率就可以计算出SNR。从而，将抽象信道建模问题归结为信号功率计算问题。

- 经验性方法：

  无线信道的模型通常可以表示为：

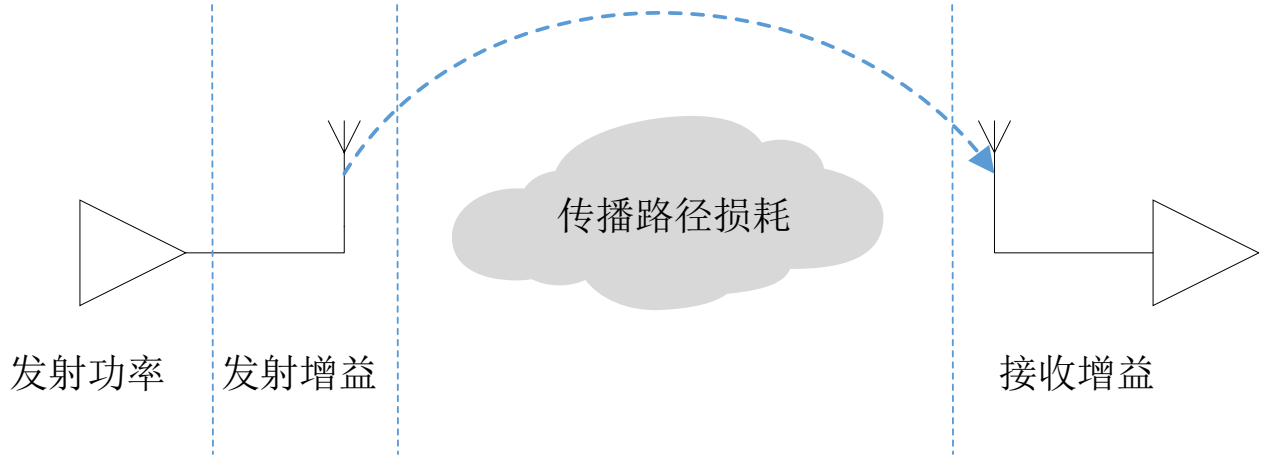
$$h^2 = h_{\text{pl}}^2 \cdot h_{\text{sh}}^2 \cdot h_{\text{fad}}^2$$

  - h_{pl}^2 表示路径损耗，取决于收发之间的距离；
  - h_{sh}^2 表示阴影效应，描述了所预测路径损耗的偏差；
  - h_{fad}^2 表示多径传播中的小尺度衰落。
  - 在经验性的路径损耗方法中，对上述三个因子进行分别建模。



链路预算基本理论

- 链路预算是评估和指导无线收发能力的关键指标，
- 主要从发射机、接收机以及收发之间的信道来进行评估，包括发射功率、发射增益、路径传播损耗、接收增益等



信号接收功率 (dBm) = 发射功率 (dBm) + 发射增益 (dB) − 路径损耗 (dB) + 接收增益 (dB)

- 发射增益一般指发射天线增益
- 接收增益一般指接收天线增益

- 获得接收信号功率的最终目的在于获得**接收信噪比**
- **接收信噪比（dB） = 接收信号功率 (dBm) – 噪声功率（dBm)**

信道模型建模

5.1、信道建模基本理论

5.2、在NS-3中实现信道建模

（1）路径损耗模型

- **Okumura模型**： Okumura模型适用于频率在150MHz至3GHz范围内、距发射天线1km至100km的城市场景。

$$h_{pl}^2[dB] = h_{pl,\,free}^2 + A_{mu}(f,d) - G(h_{te}) - G(h_{re}) - G_{A_{rea}}$$

- $h_{pl,\,free}^2$：自由空间传播损耗
- A_{mu}：相对于自由空间的中值衰减
- $G(h_{te})$：基站天线增益
- $G(h_{re})$：移动天线增益
- $G_{A_{rea}}$：基于地形类型的增益

- **Hata模型：** Hata模型代表Okumura模型的经验性公式

$$h_{pl}^2[dB] = 69.55 + 26.16\log f_c - 13.82\log h_{te} + (44.9 - 6.55\log h_{te})\log d - a(h_{re})$$

- f_c：频率
- h_{te}：基站的高度
- h_{re}：接收机天线的高度
- $a(h_{re})$：有效移动天线高度的校正因子
- d：接收器与基站的距离

（1）路径损耗模型

- **COST231 Hata模型：**
  - COST工作委员会开发的一种无线电传播模型，扩展了Hata模型，以覆盖更精细的频率范围
  - 该模型适用于城市地区，并进一步评估郊区或农村准开放/开放区域的路径损耗。
  - 适用于频率在1500MHz至2GHz范围，小区半径大于1km的系统。
  - 发射有效天线高度：30-200米之间
  - 接收有效天线高度：1-10米之间
  - 路径损耗计算的经验公式：

$$L = 46.3 + 33.9\lg f - 13.82\lg h_b - a(h_m) + (44.9 - 6.55\lg h_b)\lg d + C_m$$

  - d：单位km
  - f：单位MHz
  - h_b、h_m：基站和移动台天线有效高度，单位m
  - $a(hm)$：移动天线高度修正因子
  - C_m：城市修正因子

$$C_m = \begin{cases} 0\ dB, & \text{中等城市和郊区} \\ 3\ dB, & \text{大城市中心} \end{cases}$$

（1）路径损耗模型

- **Okumura Hata &COST231 Hata模型建模：**

```
double fmhz = m_frequency / 1e6;
double dist = a->GetDistanceFrom (b) / 1000.0; //接收器与基站之间的距离
if (m_frequency <= 1.500e9) //通过频率判断使用Okumura-Hata模型还是COST231 Hata模型
  {
    // Okumura-Hata模型
    double log_f = std::log10 (fmhz);//对频率进行对数化
    //基站的高度
    double hb = (a->GetPosition ().z > b->GetPosition ().z ? a->GetPosition ().z : b->GetPosition ().z);
    //接收机天线的高度
    double hm = (a->GetPosition ().z < b->GetPosition ().z ? a->GetPosition ().z : b->GetPosition ().z);
    NS_ASSERT_MSG (hb > 0 && hm > 0, "nodes' height must be greater then 0");
    double log_aHeight = 13.82 * std::log10 (hb);
    double log_bHeight = 0.0;
    //通过城市大小计算 有效移动天线高度的校正因子 a（hre）
    if (m_citySize == LargeCity)
      {if (fmhz < 200)
          {log_bHeight = 8.29 * std::pow (log10 (1.54 * hm), 2) -  1.1;}
       else {log_bHeight = 3.2 * std::pow (log10 (11.75 * hm), 2) - 4.97;}}
    else {log_bHeight = 0.8 + (1.1 * log_f - 0.7) * hm - 1.56 * log_f;}
```

（1）路径损耗模型

- **Okumura Hata &COST231 Hata模型建模：**

$$h_{pl}^2[dB] = 69.55 + 26.16\log f_c - 13.82\log h_{te} + (44.9 - 6.55\log h_{te})\log d - a(h_{re})$$

```
//计算路径损耗值
loss =
69.55 + (26.16 * log_f) - log_aHeight + (((44.9 - (6.55 * std::log10 (hb)) )) * std::log10 (dist)) - log_bHeight;
```

（1）路径损耗模型

- **Okumura Hata &COST231 Hata模型建模：**

```
// COST231 Hata model
double log_f = std::log10 (fmhz); //对频率进行对数化
//基站的高度
double hb = (a->GetPosition ().z > b->GetPosition ().z ? a->GetPosition ().z : b->GetPosition ().z);
//接收机的高度
double hm = (a->GetPosition ().z < b->GetPosition ().z ? a->GetPosition ().z : b->GetPosition ().z);
NS_ASSERT_MSG (hb > 0 && hm > 0, "nodes' height must be greater then 0");
double log_aHeight = 13.82 * std::log10 (hb);
double log_bHeight = 0.0;
double C = 0.0;
//根据城市大小计算移动天线高度的修正因子
if (m_citySize == LargeCity)
  {log_bHeight = 3.2 * std::pow ((std::log10 (11.75 * hm)), 2);
    C = 3; //大城市中的C为3dB }
else { log_bHeight = (1.1 * log_f - 0.7) * hm - (1.56 * log_f - 0.8); }
//计算路径损耗值
loss =
46.3 + (33.9 * log_f) - log_aHeight + (((44.9 - (6.55 * std::log10 (hb)) )) * std::log10 (dist)) - log_bHeight + C;
```

$$L = 46.3 + 33.9\lg f - 13.82\lg h_b + (44.9 - 6.55\lg h_b)\lg d + C_m - a(h_m)$$

（1）路径损耗模型

- **Walfisch-Ikegami模型**：Okumura-Hata模型没有明确地模拟屋顶衍射效应，Walfisch-Ikegami模型考虑了这些效应
  - 在非视距情况下，该模型为：

$$h_{pl}^{2}[dB] = h_{pl,\,free}^{2} + L_{rts} + L_{ms}$$

  - $h_{pl,\,free}^{2}$ ：自由空间路径损耗
  - L_{rts} ：屋顶到街道的衍射项 $L_{rts}[dB] = -16.9 - 10\log\frac{w}{\mathrm{m}} + 10\log\frac{f}{\mathrm{Hz}} + 20\log\frac{\Delta h_{Mobile}}{\mathrm{m}} + L_{Ori}$
  - w ：道路的平均宽度
  - f ：频率
  - Δh_{Mobile} ：发送者和接收者之间的高度差
  - L_{Ori} ：街道的方向

（2）随机阴影模型

- 相距 r 的两点之间的阴影自相关指数模型为：

$$\rho(r)=\frac{1}{\sigma^2}e^{\frac{r}{r_C}}$$

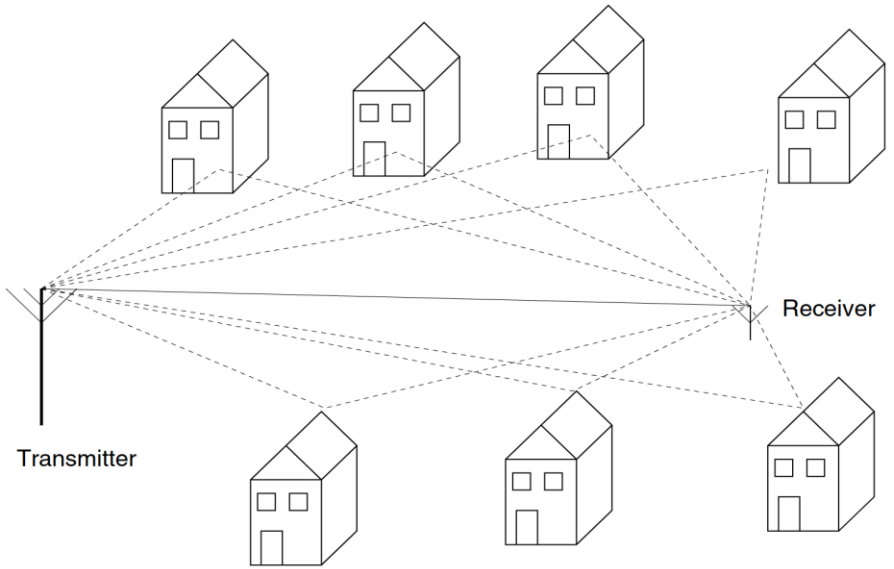
  - r_C：两个点的相关距离：1900MHz时在25米至100米之间变化，900MHz时在几米至几十米之间变化。

- 模拟某位置 b 处的阴影增益 $h^2_{\mathrm{sh},b}$（已知该位置与阴影增益为 $h^2_{\mathrm{sh},a}$ 的 a 处相隔 r 米），可以得到

$$h^2_{\mathrm{sh},b}[dB]=\rho(r)h^2_{\mathrm{sh},a}+\sqrt{1-\rho^2(r)}X$$

其中，随机变量 X 为高斯随机变量 $N(0,\sigma)$。

（3）随机衰落模型



- 衰落：建模为时间和频率的随机过程
- 瑞利分布：瑞利分布对应于具有NLOS场景的传播环境，例如室内场景，以及城市地区的宏蜂窝

$$p(h_{\text{fad}}) = h_{\text{fad}} \cdot e^{\frac{-h_{\text{fad}}^{\,2}}{2}}$$

BuildingsPropagationLossModel：阴影衰落

- ShadowSigmaOutdoor：
  - 用于计算室外节点阴影所使用正态分布的标准差
  - 初始值为7
- ShadowSigmaIndoor：
  - 用于计算室内节点阴影所使用正态分布的标准差
  - 初始值为8
- ShadowSigmaExtWalls：
  - 用于计算由于外墙引起阴影所使用正态分布的标准差
  - 初始值为5

（3）随机衰落模型

- 莱斯分布：接收端存在一个主要的静态信号时，如LOS分量

$$p(R) = \frac{R}{\sigma^2} exp\left(-\frac{R^2 + A^2}{2\sigma^2}\right) I_0(\frac{RA}{\sigma^2})$$

  - R：正弦信号加窄带高斯随机变量的包络
  - A：主信号幅度的峰值
  - σ^2：多径信号分量的功率
  - $I_0()$：修正的0阶第一类贝塞尔函数

（3）随机衰落模型

- Nakagami-m分布：比瑞利分布、莱斯分布更好地接近实验测试数据

$$p(r)=\frac{2m^{m}r^{2m-1}}{\Omega^{m}\Gamma(m)}exp\left(-\frac{mr^{2}}{\Omega}\right),\qquad m\geq 1/2, r\geq 0$$

  - Ω：多径散射场的平均功率
  - m：Nakagami的形状因子，描述由于多径效应引起的衰落程度
    - m=1/2：Nakagami分布变为单边高斯分布
    - m=1：Nakagami分布变为瑞利分布
    - m>1：Nakagami分布接近莱斯分布

（5）NakagamiPropagationLossModel

- Nakagami-m快速衰落传播损耗模型：
  - 概率密度函数（m是衰落深度参数，ω是平均接收功率）：$p(x;m;\omega)=\frac{2m^m}{\Gamma(m)\omega^m}x^{2m-1}e^{-\frac{m}{\omega}x^2}$
  - 允许为三个不同的距离范围指定不同的m参数：

$$\underbrace{0\ldots\ldots d_1}_{m_0}\ \underbrace{.\ldots\ldots d_2}_{m_1}\ \underbrace{.\ldots\ldots d_3}_{m_2}$$

  - m = 1：Nakagami-m 分布等于瑞利分布
- NakagamiPropagationLoss模型的主要属性：
  - Distance1：第二距离场的开始。默认值为80米。
  - Distance2：第三距离场的开始。默认值为200米。
  - m0：m0表示距离小于Distance1的距离。默认值为1.5。
  - m1：m1表示距离小于Distance2的距离。默认值为0.75。
  - m2：m2的距离大于Distance2。默认值为0.75。
  - BSAntennaHeight：BS天线高度（默认为50米）。
  - SSAntennaHeight：SS天线高度（默认为3米）。
  - MinDistance：传播模型无法给出结果的距离（m）。

THANKS FOR WATCHING