浙江大学第十五届
大学生数学建模竞赛

2017 年 5 月 2 日 - 5 月 12 日

编号　　796

题目　　A✓　　B

（在所选题目上打勾）

	参赛队员 1	参赛队员 2	参赛队员 3
姓名	何康瑞	蒋文	曹哲锋
学号	3160103786	3160101006	3160101534
院（系）	求是学院	求是学院	求是学院
专业	工科试验班（信息）	工科试验班（信息）	飞行器设计与工程
手机	13958868651	13306519036	13754322264
Email	472444974@qq.com	Wenjiang.wj@foxmail.com	1241582721@qq.com

浙江大学本科生院

浙江大学数学建模实践基地

目录

摘要：

本文围绕网络侧估计终端用户视频体验问题进行讨论，建立了基于遗传算法的函数关系求解模型和基于模糊聚类分析的变量分离函数关系求解模型两种数学模型。利用现代生物进化理论和遗传学知识构建出的基于本题目深度优化的多基因变量，动态筛选原则的遗传算法，利用模糊聚类法处理本题目数目庞大的数据。

基于遗传算法的函数关系求解模型：首先，对于题设数据进行了无量纲化处理，采用 MinMax 归一化方式对原始数据进行线性变化，方便后期处理。其次，在对数据进行分析的基础上建立了多参数的目标函数模型，并进行初步计算确定参数预期范围。再次，根据参数范围建立了固定编码长度的遗传因子解编码算法，设定了适应度函数计算适应度累积量，再以此为根据进行自然选择。经过自然选择的个体通过基于基因突变与基因重组过程将基因遗传给下一代，进行多次迭代后收敛于最优解从而得出系数。

在模型求解中，通常于 300 代内实现基因频率的收敛，随着迭代次数的增加，关于初始缓冲时延的目标函数在反复随机抽取 800 个测试样本中的累积适应度积累量由 35.7356～490.4525 收敛到 8.9936～9.0065 的理想最优值。关于卡顿占比的目标函数在同样的测试手段中也有初始的 39.697695～625.799693 收敛于 2.298186～2.313633 的理想估计值，函数关系模型基本达成了目标。

基于模糊聚类分析的变量分离函数关系求解模型:首先，对于题设数据进行无量纲化处理，采用 MinMax 归一化对原始数据进行线性变化，方便后期处理。其次，为了达到在处理一个侧变量与评价变量之间的函数关系的同时保证其余两个参数大致保持不变的目的，采用模糊聚类法，确定截取水平 λ 为 0.2，进而将数据进行模糊聚类，将聚类后的数据各自分为五组，两两组合，组合出符合不同条件的样本后对数据进行二次多项式的拟合，进而得出相应的分段函数。最后综合这些分段函数即为所求的函数关系模型。

本文在最后给出了针对模型优缺点的评价，并针对模型特点提出了改进方向。

关键词：网络侧估计终端用户视频；遗传算法；现代生物进化理论；模糊聚类分析

1. 问题重述

随着无线宽带网络的升级，以及智能终端的普及，越来越多的用户选择在移动智能终端上用应用客户端 APP 观看网络视频，这是一种基于 TCP 的视频传输及播放。看网络视频影响用户体验的两个关键指标是初始缓冲等待时间和在视频播放过程中的卡顿缓冲时间，我们可以用初始缓冲时延和卡顿时长占比（卡顿时长占比= 卡顿时长/ 视频播放时长）来定量评价用户体验。研究表明影响初始缓冲时延和卡顿时长占比的主要因素有初始缓冲峰值速率、播放阶段平均下载速率、端到端环回时间（E2E RTT），以及视频参数。然而这些因素和初始 缓冲时延和卡顿时长占比之间的关系并不明确。

试根据附件提供的实验数据建立用户体验评价变量（初始缓冲时延，卡顿时长占比）与网络侧变量（初始缓冲峰值速率，播放阶段平均下载速率，E2E RTT）之间的函数关系。

2 问题的分析

这道题我们建立了两种解题模型。

2.1 按遗传算法分析数据

本题要求我们寻找用户侧与网络侧间的函数关系，我们先对题设数据进行初步的计算与分析，以网络侧变量为因变量，用户体验指数为自变量设计预期的目标函数模型，再利用自行实现的基于该题目深度定制的多基因，多进化方向，动态筛选标准的遗传算法求解最优的参数与参数组合。在具体的实现过程中自行设置了针对该题目特殊定制的遗传密码解编码方式，使得基因突变对于个体表现型的差异影响控制在预期最优解范围内，参照原核生物的基因重组规律对对每代的部分个体基因进行定向组合。采用大样本种群数量与动态随机抽样检验样本相结合的方式，尽可能的避免函数出现过拟合与陷入求取局部最优解的常见困境。在数百代迭代后种群的基因频率变为一稳定值，此时即为预计求取的最优解。

2.2 利用模糊聚类分析数据

为了得到用户体验评价变量（初始缓冲时延，卡顿时长占比）与网络侧变量（初始缓冲峰值速率，播放阶段平均下载速率，E2E RTT）之间的函数关系，我们处理了附件中给出的相关数据。由于在MATLAB中只支持拟合两个变量之间的函数关系，而仅仅单独地得得到单个网络侧变量与评价变量的函数关系显然不符合真实情况。所以为了在考虑一个网络侧变量的同时兼顾其余两个网络侧变量，我们采用了模糊聚类法，在处理一个侧变量与评价变量之间的函数关系的同时保证其余两个参数大致保持不变，筛选出符合条件的样本后得出相应的分段函数。最后综合这些分段函数即为所求的函数关系。

3 符号约定

符号	意义
α	基因突变概率
β	基因重组概率
$X_{i\,max}$	第 i 个自变量的最大值
$X_{i\,min}$	第 i 个自变量的最小值
N	种群个数
R_i	第 i 个个体的适应能力在种群中的排名
P_i	第 i 个个体在种群中的生存概率
F_1	该函数对第一个变量的拟合度
$\bar{y}$	在该测试样本中应变量的实际值

4 模型一的准备：

4.1 数据的预处理：

结合数据的特点，我们采取 Min-Max 归一化手段将原始数据做一线性变换映射到[0, 1] 区间内，并以此方法去除数据量纲。具体的数据归一化与反归一化处理方案如下

$$\begin{cases} X' = \dfrac{X - X_{min}}{X_{max} - X_{min}} & X = (X_{max} - X_{min})X' + X_{min} \\ Y' = \dfrac{Y - Y_{min}}{Y_{max} - Y_{min}} & Y = (Y_{max} - Y_{min})Y' + Y_{min} \end{cases}$$

4.2 寻找目标函数公式基本形式

由预先观察分析数据时所得出的结论，我们设立了满足指数，对数，倒数以及多项式方程的目标函数模型，再求解出其中的参数。

$$\begin{aligned} y = {} & a_1 e^{a_2 x_1} + a_3 x_1^2 + a_4 x_1 + a_5 \frac{1}{1+x_1} + a_6 ln(1+x_1) \\ & + b_1 e^{b_2 x_2} + b_3 x_2^2 + b_4 x_2 + b_5 \frac{1}{1+x_2} + b_6 ln(1+x_2) \\ & + c_1 e^{c_2 x_3} + c_3 x_3^2 + c_4 x_3 + c_5 \frac{1}{1+x_3} + c_6 ln(1+x_3) + C \end{aligned}$$

4.3 简述遗传算法

遗传算法是基于现代生物进化理论与遗传学知识的求解最优化问题的自适应性人工智能算法[1]。利用计算机模拟生物学中遗传物质的表达与复制，等位基因的突变与重组，自然环境对物种表现型的定向筛选实现对基因频率的间接影响。遗传算法是具有普适性的优化问题解决框架。目前遗传算法已成功应用到许多领域，如优化设计[2]、神经网络训练、模式识别、时序预测等.

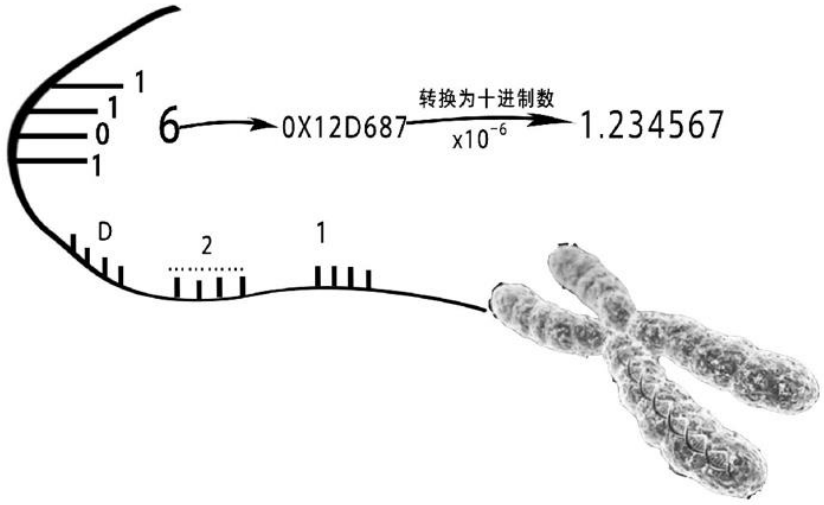
在 MATLAB 中有关于遗传算法的工具箱，但是由于我们解决函数对应关系的问题的复杂性不便于直接调用工具箱，以及基于针对问题模型对遗传算法深度优化定制的原则我们用 C++重新实现一个功能及模拟方式与 MATLAB 多有不同的遗传算法。(注： 支撑材料中

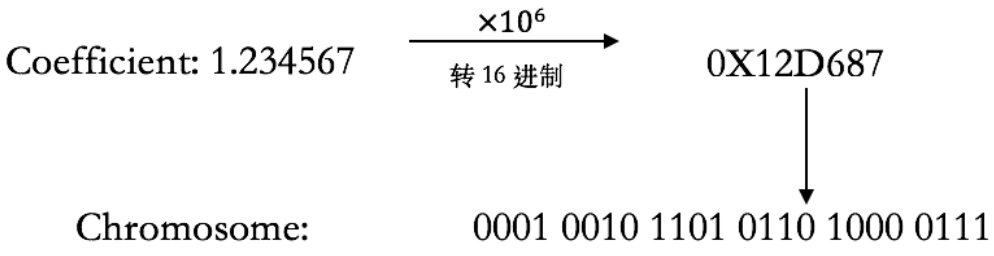
的 GA.cpp 即为遗传算法的主体实现部分，编译平台: Mac OS Sierra 10.12.4 编译器: Apple LLVM version 8.1.0 (clang-802.0.42) 语言标准 C++ 11 编译选项： -O3 -std=C++11)

5 模型一的建立

5.1 遗传基因解编码算法的确定

出于计算复杂度的要求，我们并没有直接套用现实生活中四种碱基组成遗传密码的编码方式，而是采用 01 字符串这种对计算机更为友好的方式作为遗传密码。对初期设计的函数模型进行预计算，考虑到我们使用的以归一化的数据只会出现在 0～1 区间内，所以我们的参数的预期合理范围应在-10～+10 之间，在考虑到计算机中双精度浮点数的精确程度，我们采取选取 1 位个位数和 6 位小数转化为 7 位整数。再将整数转变为 7 个 16 进制数，利用单个十六进制数与 4 位二进制数的映射关系得到 01 字符串编码。





需要注意的是该编码方式与直接将整数转变为二进制数不同，而是一种我们自行设计的 Hash 算法，该编码方式的特点是解编码的位数恒定，使得基因突变后得到的系数认为处在我们认为的合理范围之内。对应到现代生物进化理论中的基因突变可以理解为合理的基因突变才可产生能合理生存的子代。

与编码方式相似，利用同等的逆变换后在转化为小数即为得到的方程系数，即遗传因子解码，表达为个体形状的过程。

5.2 个体基因初始值的确定

考虑到此前对于合理系数的出现范围的预期，我们对于每一系数均以范围内随机数生成的方法进行求取，考虑到我们的种群个数相对系数区间而言求取的极大 ($X_{max} = 10000$)，我们所得的个体系数分布将极其稠密，从而尽可能避免只求得局部最优解的情况。

5.2.1 个体适应度计算

以归一化后的题设数据为标准，模拟自然环境进行对种群进行筛选。我们以个体为研究对象出发，对每个个体的环境适应程度进行计算。计算的方式是利用适应度函数计算每一个体染色体中所携带的系数方程带入实际样本中自变量所得的因变量值与实际样本中因变量值所得差的绝对值进行求和，累加后得到得值即为该个体的适应度积累量。

$$F_1 = \sum_{i=1}^{N} \left| f\left(x_{1i}, x_{2i}, x_{3i}\right) - \bar{y_{1i}} \right|$$

在这当中有两点值得考虑，其一计算适应度是不采用方差。由于去量纲化后的数据统一映射在[0, 1] 区间上，所有计算出的偏差也普遍小于 1，平方累积后个体差异减小，不利于优势基因的保留与弱势基因的淘汰。

其二，为了防止对于数据的过拟合，以及出于尽可能模拟自然选择情况的原则，我们每次用于检测适应对的测试样本为从题设的 89266 条数据抽取 800 条作为验证样本，体现出筛选环境的多边形，从而增强了模型的可靠性。考虑到我们的种群基因收敛代数在 100 这个数量级，每次检测时抽取不到 1%的数据是合理的。

5.2.2 个体筛选概率的确定

在确定了每个个体的适应程度进行计算后我们再依照适应程度对个体进行排名，个体适应度越高，其适应度积累量越低，也就是说其染色体所对应系数组成的函数越符合随机抽取的测试样本。

表 1: 最优劣个体适应对比图

迭代的次数	最优个体适应度累计量	最劣个体适应度累计量
1	35.7356	490.4525
2	71.5323	2449.4034
3	53.4225	908.4149
4	51.3486	1714.8254
5	60.6693	496.1959
……	……	……
398	5.6792	5.7165
399	7.4337	7.4592
400	8.9936	9.0050

然后我们对适应度累积量由低到高的个体依次赋以生存概率，为便于进行下一代的基因增殖与重组，维持种群总数稳定，我们每次筛选出一半的幸存个体，考虑到积累量排名与生存概率的正相关关系以及筛选一半种群个数的目标，我们得到生存概率的计算公式如下：

$$P(\xi) = 1 - \frac{1}{N} r_i$$

再根据其生存概率进行“轮盘赌”后筛选出存活个体，可以计算出幸存个体的数目的数学期望。

$$E(\xi) = \sum_{i=1}^{N} P_i$$

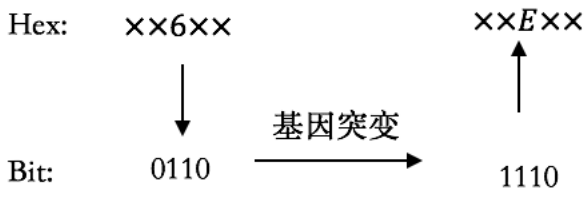
对于随机筛选后的结果不严格等于种群数目一半的情况，我们可以依据个体对适应度累积量的高低来进行细微的调整使得其幸存个体数保持为一半，幸存个体并非永生，而是拥有将基因遗传给下一代的资格。

5.2.3 个体基因的复制与突变

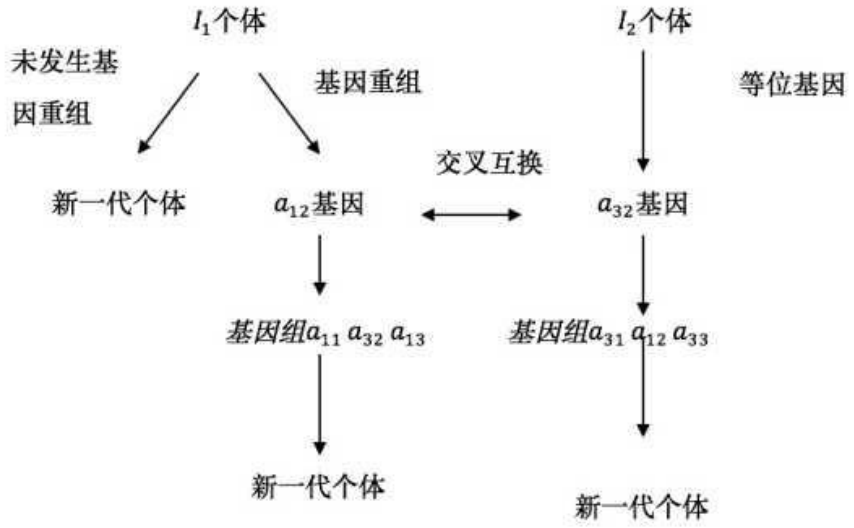
对幸存的染色体进行复制与加倍便可得到遗传给下一代的遗传物质，在生物学中的此过程相应为 DNA 的复制，由于 DNA 双链解螺旋，其碱基稳定性下降，是基因突变发生的主要时期。

在这一时期当中我们仍然采依据概率进行随机判定的方法在每对基因的层面上对是否突变进行选择。参照基因突变的系数 $\alpha = 0.001$， 对于随机选取得到的基因发生基因突变的节点位置进行随机确定。出于二进制字符串的编码特点，我们只需将发生基因突变的节点进行取反运算即可。

以单个基因为随机概率计算单位，以改变遗传密码上某一位的二进制值为突变目标，实现了对真实情况的最大化模拟。



5.2.4 基因重组

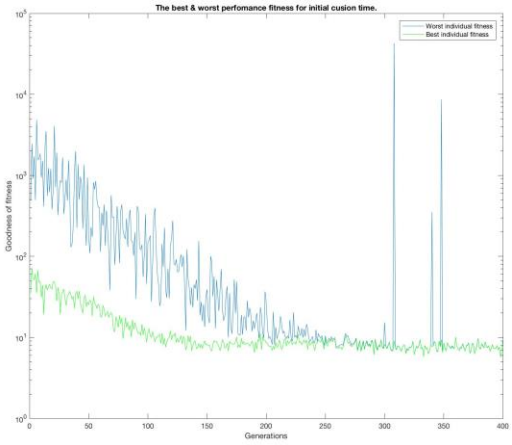


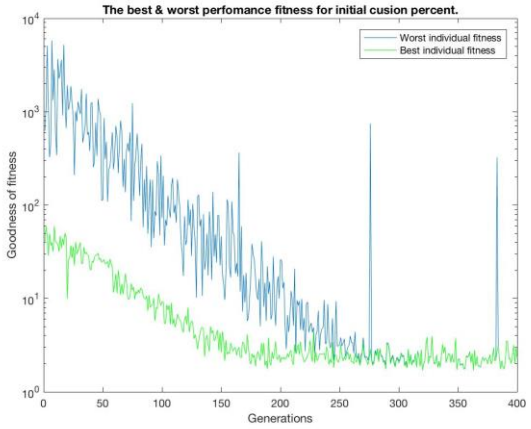
在通过染色体的复制得到了与种群最大个数相等的染色体后我们考虑等位基因间的基因重组。我们选为参照的是原核生物（以大肠杆菌）这类无性生殖的个体的基因重组时的染色体行为。同样以基因重组概率 β = 0.5 的随机方式进行确定基因是否发生与等位基因的重组，当发生基因重组时，该基因与种群中随机一个个体的染色体上的等位基因进行互换，从而保证了个体形状的多样性，更有可能选取出具有最优解的系数组合。

5.2.5 种群的迭代

上述 3.3 过程为一代以内个体适应度的确定，基因留存状态的基于概率随机确定，基因突变与加倍，基因重组的基于概率的随机判定。当我们对每一代进行统一模式的定向选择与遗传后，我们记录下每次表现最优个体和表现最劣个体的适应度累积量在程序中进行实时显示，客观的表现出目前种群的进化情况与种群基因频率的收敛程度。程序每循环 100 代后停止等待外界输入指令是否继续迭代，实现对遗传算法的动态决策。

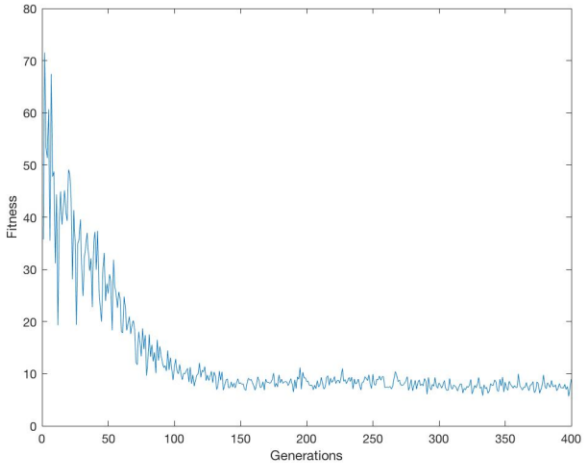
运行程序后我们可以看到初期最优个体与最劣个体之间差距极大，多代筛选后可以发现种群整体基因型向着适应度累积量减小的方向进化，但是由于基因突变和环境（即从原始数据中随机的选取检验样本）的变化，整体呈现波动下降的下降，最劣个体表现受非良性基因突变的影响会出现短暂的剧烈上升现象，等很快便会被自然选择淘汰，经过数百代

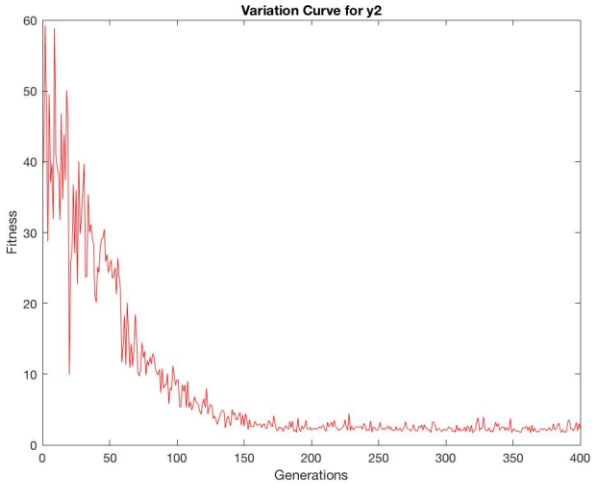




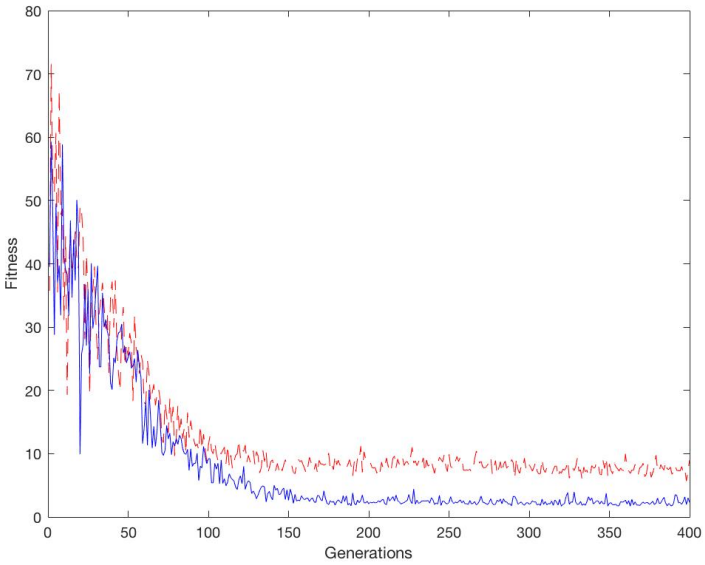
（普遍为 300 代左右，受初始随机值的影响有所波动）后种群基因会呈现高度的同质化，数据上也收敛于整体的最优值，即找到了该函数模型下的系数最优值与最优组合。

单独考虑最优基因表现情况可以发现由于我们使用的遗传算法与普遍使用的单一变量单一适应度标准的遗传算法的差别，种群最优基因个体的适应度累积量呈现阶梯式波动下降的特点，由于系数变量的多元性，基因突变的效果具有累积性与滞后性。也就是说，普遍会出现多个系数基因出现基因突变后才能给个体适应度累积量带来明显的数量级上的变化。





5.2.6 模型的推广



由于该算法求取的是任意自变量与因变量之间的函数关系的系数最优解，当我们把算法读入的测试样本因变量设置为不同变量时，便可根据自变量得出针对不同变量间的函数关系。当然，我们也可以看到不同因变量间于我们所给定的函数的差异造成最优个体的适应度累积量的差异性。

计算结果的得出：根据以上算法步骤，我们对数学模型进行编程实现后选取基因收敛后的最优个体的染色体进行解码后得到系数向量为 Y1 Y2

将最优解系数代入预先规划的函数系数当中，可得无量纲化的自变量初始缓冲峰值速率(kbps) E2E RTT(ms) 播放阶段平均速率(kbps) 与无量纲化的因变量初始缓冲时延(ms) 卡顿占比之间的函数关系.

再对自变量与因变量做去量纲化时的线性变换的逆变换后即可得到含量纲的函数关系解：

$$
\begin{aligned}
y_1 = {} & 978e^{0.02-0.0000625x_1} + 1.17\cdot 10^{-7}x_1^2 - 0.013x_1 - \frac{2.1\cdot 10^8}{158617+x_1} + 2324.6ln(\frac{158617+x_1}{158921}) \\
& - 2337e^{0.22-0.015x_2} + 0.13x_2^2 + 6.73x_2 + \frac{74250}{119+x_2} - 4380ln(\frac{119+x_2}{134}) \\
& + 4061e^{0.0006-0.00006x_3} + 5.68\cdot 10^{-7}x_3^2 + 0.088x_3 + \frac{4.7\cdot 10^7}{29312+x_3} + 4157ln(\frac{29312+x_3}{29322}) + 261
\end{aligned}
$$

$$y_2 = 2.12e^{0.002-6.81\cdot10^{-6}x_1} + 1.12\cdot10^{-10}x_1^2 + 2.8\cdot10^{-6}x_1 - \frac{105583}{158617+x_1} + 0.183ln(\frac{158617+x_1}{158921})$$
$$+ 1.783e^{0.23-0.016x_2} + 0.027x_2^2 - 0.00014x_2 - \frac{287}{119+x_2} - 0.73ln(\frac{119+x_2}{134})$$
$$- 4.22e^{0.0005-0.00005x_3} - 1.66\cdot10^{-8}x_3^2 + 0.0002888x_3 + \frac{13942}{29312+x_3} + 11.5ln(\frac{29312+x_3}{29322}) + 0.32$$

6 模型的评估与改进

该数学模型的特点是采取基于对该问题深度优化定制的多基因变量，动态选择标准的遗传算法的实现，利用大种群数目保证系数解在预期解区间内的稠密性找到，函数关系的系数的最优解。通过该方法的寻找到的函数关系的特点在于函数拟合效果好，经历多代筛选后的拟合函数与八百个测试样本的绝对偏差的累积值对变量 y1、y2 分别在 8.9 与 2.3 以内，即单个测试点的平均偏差不超过 0.011125 和 0.002875 同时由于每次采取的筛选环境由随机抽取的 800 个测试样本组成，在一定程度上避免了函数的过拟合。自定义的基因解编码方式使得基因突变后的解仍能在预期最优范围之内。

该模型可优化改进的地方有基因表达方式中显隐形基因的表达规则，加入显隐性基因的表达特征后可以有效的保存暂时不利于环境的突变，等到其他系数基因也发生这类突变进行累积后这些基因一直表达得到“超级个体”，这也是生物进化中常出现的现象。其次是函数模型的再优化，在本次数学模型中我们针对两因变量使用同一函数模型进行系数求解，可以看到我们最终得到的方程中的适应度有较大的差距，虽然这次利用了我们的解题模型的通用性，但是如果根据不同变量设置不同的目标函数也许所得方程的拟合程度会有更大的提升。

7 模型二的准备

7.1 模型二的假设

（1）网络侧变量（初始缓冲峰值速率，播放阶段平均下载速率，E2E RTT）之间存在相互关系，对其他侧变量与评价变量（初始缓冲时延，卡顿时长占比）之间的函数关系存在一定的影响。

（2）网络侧变量在相近的一定区间范围内对其余侧变量和评价变量的函数关系的影响相同。

（3）假定网络侧变量与评价变量之间的关系在一定区间内符合2次多项式。

7.2 模型二的预处理

首先为了将数据更好的分类处理，我们将初始缓冲峰值速率，播放阶段平均下载速率，E2E RTT，初始缓冲时延，卡顿时长占比这五组数据分别进行归一化处理。结合数据的特点，我们采取Min-Max归一化手段将原始数据做一线性变换映射到[0, 1] 区间内，并以此方法去除数据量纲。

7.3 模糊类聚

所谓模糊聚类分析，一般是指根据研究对象本身的属性来构造模糊矩阵，并在此基础上根据一定的隶属度来确定聚类关系，即用模糊数学的方法把样本之间的模糊关系定量的确定，从而客观且准确地进行聚类。聚类就是将数据集分成多个类或簇，使得各个类之间的数据差别应尽可能大，类内之间的数据差别应尽可能小，即为“最小化类间相似性，最大化类内相似性”原则。

这里我们确定截取水平 λ 为0.2，将五组归一化在0~1下的数据,按 λ 各自分为5组，并按照两两组合分为75组数据。

8 模型二的建立

8.1 函数拟合

在处理与一个网络侧变量与评价变量之间的函数关系的时候，我们将其他两个侧变量分别控制大致区间相同，两两组合，总共分为25组，即拟合25组多项式的系数。

表 2: 平均速率-延时

系数a	系数b	系数c	初始缓冲速率区间	E2E RTT区间
-0.9417	0.3908	-0.0360	0.8 ~1	0 ~0.2
-0.1620	0.1952	-0.0006	0.4 ~0.6	0.6 ~0.8
-0.1159	0.1235	0.0110	0.2 ~0.4	0.4 ~0.6
-0.0304	0.0172	0.0017	0.6 ~0.8	0 ~0.2
0.0000	0.0000	0.0000	0.6 ~0.8	0.6 ~0.8
0.0000	0.0000	0.0000	0.6 ~0.8	0.8 ~1
0.0000	0.9830	-0.1801	0.8 ~1	0.4 ~0.6
0.0000	0.0000	0.0000	0.8 ~1	0.6 ~0.8
0.0000	0.0000	0.0000	0.8 ~1	0.8 ~1
0.0029	-0.0018	0.0187	0.2 ~0.4	0.2 ~0.4
0.0037	0.0280	0.0044	0.2 ~0.4	0 ~0.2
0.0104	0.0176	0.0021	0.4 ~0.6	0 ~0.2
0.0156	0.0103	0.0085	0.6 ~0.8	0.2 ~0.4
0.0622	-0.0337	0.0208	0.6 ~0.8	0.4 ~0.6
0.0671	-0.0029	0.0121	0.4 ~0.6	0.2 ~0.4
0.2249	-0.0126	0.0167	0.4 ~0.6	0.4 ~0.6
0.3634	-0.0448	0.0394	0.2 ~0.4	0.6 ~0.8
0.4863	-0.1849	0.0739	0.2 ~0.4	0.8 ~1
1.7261	-1.0147	0.1592	0 ~0.2	0 ~0.2
4.9408	-2.2904	0.2977	0 ~0.2	0.2 ~0.4
5.0407	-2.1446	0.3274	0 ~0.2	0.8 ~1
5.4924	-2.2277	0.2347	0.8 ~1	0.2 ~0.4
6.7981	-3.0003	0.3992	0 ~0.2	0.6 ~0.8
8.4065	-3.4164	0.3993	0 ~0.2	0.4 ~0.6
28.8452	-12.1430	1.3204	0.4 ~0.6	0.8 ~1

表 3: 平均速率-卡顿占比

系数a	系数b	系数c	初始缓冲速率区间	E2E RTT区间
0.0000	0.0000	0.0000	0.4 ~0.6	0.8 ~1
0.0000	0.0000	0.0000	0.6 ~0.8	0.6 ~0.8
0.0000	0.0000	0.0000	0.6 ~0.8	0.8 ~1
0.0000	0.0000	0.0000	0.8 ~1	0 ~0.2
0.0000	0.0000	0.0000	0.8 ~1	0.2 ~0.4
0.0000	0.0000	0.0000	0.8 ~1	0.4 ~0.6
0.0000	0.0000	0.0000	0.8 ~1	0.6 ~0.8
0.0000	0.0000	0.0000	0.8 ~1	0.8 ~1
0.0008	0.0003	0.0000	0.4 ~0.6	0.4 ~0.6
0.0021	0.0008	0.0001	0.4 ~0.6	0.6 ~0.8
0.0024	0.0010	0.0001	0.4 ~0.6	0 ~0.2
0.0072	0.0025	0.0002	0.6 ~0.8	0.2 ~0.4
0.0085	0.0030	0.0003	0.6 ~0.8	0.4 ~0.6
0.0104	0.0038	0.0003	0.4 ~0.6	0.2 ~0.4
0.0154	0.0089	0.0012	0.2 ~0.4	0.2 ~0.4
0.0167	0.0062	0.0006	0.6 ~0.8	0 ~0.2
0.0279	0.0174	0.0025	0.2 ~0.4	0.4 ~0.6
0.0458	0.0217	0.0025	0.2 ~0.4	0 ~0.2
0.0913	0.0425	0.0049	0.2 ~0.4	0.6 ~0.8
0.3299	0.1376	0.0142	0.2 ~0.4	0.8 ~1
1.6320	0.6617	0.0659	0 ~0.2	0.8 ~1
1.9390	0.7637	0.0742	0 ~0.2	0.6 ~0.8
2.1286	0.8727	0.0881	0 ~0.2	0.2 ~0.4
2.5411	0.9651	0.0901	0 ~0.2	0.4 ~0.6
2.5765	1.1409	0.1236	0 ~0.2	0 ~0.2

表 4: 初始速率-延时

系数a	系数b	系数c	初始缓冲速率区间	E2E RTT区间
-3.7622	4.0361	1.0359	0.4~0.6	0.4~0.6
-1.0230	0.5583	0.0584	0~0.2	0.2~0.4
-0.6463	0.5743	0.1050	0.2~0.4	0.4~0.6
-0.3524	0.6065	0.2472	0.2~0.4	0.8~1
-0.1105	0.0000	0.0862	0.4~0.6	0.6~0.8
-0.0031	0.0000	0.0100	0~0.2	0.6~0.8
0.0000	0.0000	0.0921	0~0.2	0.4~0.6
0.0000	0.0000	0.0000	0~0.2	0.8~1
0.0000	0.0000	0.0350	0.4~0.6	0.8~1
0.0000	0.0000	0.0472	0.6~0.8	0.4~0.6
0.0000	0.0000	0.0000	0.6~0.8	0.6~0.8
0.0000	0.0000	0.0425	0.6~0.8	0.8~1
0.0000	0.0000	0.0000	0.8~1	0.4~0.6
0.0000	0.0000	0.0000	0.8~1	0.6~0.8
0.0000	0.0000	0.0000	0.8~1	0.8~1
0.3847	0.1839	0.0783	0.6~0.8	0.2~0.4
0.3934	0.2451	0.0767	0.4~0.6	0.2~0.4
0.4790	0.2755	0.0615	0.2~0.4	0.2~0.4
3.9082	1.6860	0.1942	0~0.2	0~0.2
1.5986	1.0262	0.5263	0.8~1	0.2~0.4
9.6084	3.4174	0.3820	0.8~1	0~0.2
12.5230	4.4415	0.4575	0.6~0.8	0~0.2
14.1548	4.7004	0.4084	0.2~0.4	0~0.2
19.2531	6.0455	0.5003	0.4~0.6	0~0.2
20.7317	25.6918	7.9711	0.2~0.4	0.6~0.8

表 5: 初始速率-卡顿占比

系数a	系数b	系数c	初始缓冲速率区间	播放阶段平均速率区间
-15.7152	6.9968	0.7680	0.8~1	0.2~0.4
-14.3831	13.1669	2.9620	0~0.2	0.4~0.6
-11.7603	15.6627	5.1676	0.2~0.4	0.6~0.8
-9.4304	3.3681	0.2901	0.8~1	0~0.2
-4.7677	2.6690	0.3314	0.2~0.4	0.2~0.4
-2.2934	1.2960	0.1314	0~0.2	0.2~0.4
-1.1973	0.6541	0.0727	0.4~0.6	0.2~0.4
-0.4275	0.2534	0.0282	0.6~0.8	0.2~0.4
-0.3859	0.0000	0.1823	0~0.2	0.6~0.8
0.0000	0.0000	0.0000	0~0.2	0.8~1
0.0000	0.0000	0.0000	0.4~0.6	0.4~0.6
0.0000	0.0000	0.0000	0.4~0.6	0.6~0.8
0.0000	0.0000	0.0425	0.4~0.6	0.8~1
0.0000	0.0000	0.0183	0.6~0.8	0.4~0.6
0.0000	0.0000	0.0000	0.6~0.8	0.6~0.8
0.0000	0.0000	0.0000	0.6~0.8	0.8~1
0.0000	0.0000	0.0000	0.8~1	0.4~0.6
0.0000	0.0000	0.0000	0.8~1	0.6~0.8
0.0000	0.0000	0.0000	0.8~1	0.8~1
0.6076	0.2171	0.0248	0.6~0.8	0~0.2
1.1732	0.4296	0.0544	0.2~0.4	0~0.2
1.2326	1.2419	0.3340	0.2~0.4	0.4~0.6
1.3101	0.4493	0.0476	0.4~0.6	0~0.2
3.2412	5.8856	2.6823	0.2~0.4	0.8~1
12.8411	4.4224	0.4151	0~0.2	0~0.2

表 6: RTT-延时

系数a	系数b	系数c	初始缓冲速率区间	播放阶段平均速率区间
-15.7152	6.9968	0.7680	0.8~1	0.2~0.4
-14.3831	13.1669	2.9620	0~0.2	0.4~0.6
-11.7603	15.6627	5.1676	0.2~0.4	0.6~0.8
-9.4304	3.3681	0.2901	0.8~1	0~0.2
-4.7677	2.6690	0.3314	0.2~0.4	0.2~0.4
-2.2934	1.2960	0.1314	0~0.2	0.2~0.4
-1.1973	0.6541	0.0727	0.4~0.6	0.2~0.4
-0.4275	0.2534	0.0282	0.6~0.8	0.2~0.4
-0.3859	0.0000	0.1823	0~0.2	0.6~0.8
0.0000	0.0000	0.0000	0~0.2	0.8~1
0.0000	0.0000	0.0000	0.4~0.6	0.4~0.6
0.0000	0.0000	0.0000	0.4~0.6	0.6~0.8
0.0000	0.0000	0.0425	0.4~0.6	0.8~1
0.0000	0.0000	0.0183	0.6~0.8	0.4~0.6
0.0000	0.0000	0.0000	0.6~0.8	0.6~0.8
0.0000	0.0000	0.0000	0.6~0.8	0.8~1
0.0000	0.0000	0.0000	0.8~1	0.4~0.6
0.0000	0.0000	0.0000	0.8~1	0.6~0.8
0.0000	0.0000	0.0000	0.8~1	0.8~1
0.6076	0.2171	0.0248	0.6~0.8	0~0.2
1.1732	0.4296	0.0544	0.2~0.4	0~0.2
1.2326	1.2419	0.3340	0.2~0.4	0.4~0.6
1.3101	0.4493	0.0476	0.4~0.6	0~0.2
3.2412	5.8856	2.6823	0.2~0.4	0.8~1
12.8411	4.4224	0.4151	0~0.2	0~0.2

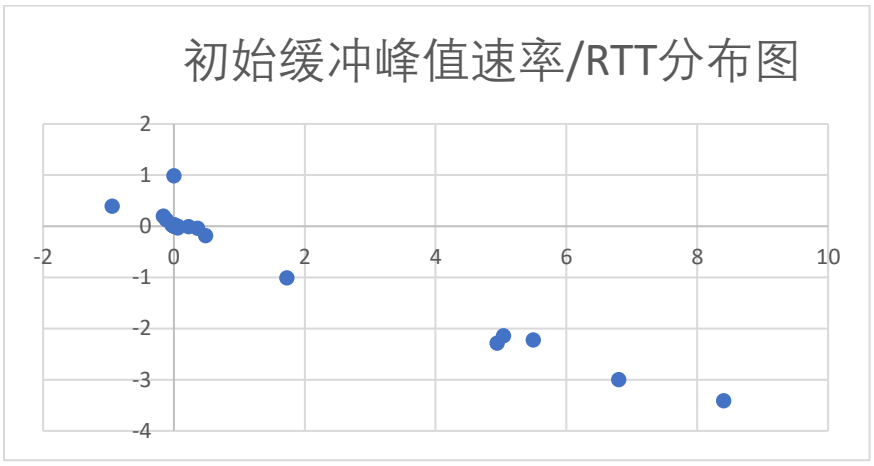
表 7: RTT-卡顿占比

系数a	系数b	系数c	初始缓冲速率区间	播放阶段平均速率区间
-0.2440	0.2230	0.0504	0~0.2	0.4~0.6
-0.1791	0.0000	0.0833	0~0.2	0.6~0.8
-0.0841	0.1376	0.0551	0.2~0.4	0.8~1
0.0000	0.0000	0.0000	0~0.2	0.8~1
0.0000	0.0000	0.0000	0.4~0.6	0.4~0.6
0.0000	0.0000	0.0000	0.4~0.6	0.6~0.8
0.0000	0.0000	0.0011	0.4~0.6	0.8~1
0.0000	0.0000	0.0000	0.6~0.8	0~0.2
0.0000	0.0000	0.0006	0.6~0.8	0.4~0.6
0.0000	0.0000	0.0000	0.6~0.8	0.6~0.8
0.0000	0.0000	0.0000	0.6~0.8	0.8~1
0.0000	0.0000	0.0000	0.8~1	0~0.2
0.0000	0.0000	0.0000	0.8~1	0.2~0.4
0.0000	0.0000	0.0000	0.8~1	0.4~0.6
0.0000	0.0000	0.0000	0.8~1	0.6~0.8
0.0000	0.0000	0.0000	0.8~1	0.8~1
0.0051	0.0015	0.0001	0.4~0.6	0.2~0.4
0.0078	0.0028	0.0003	0.4~0.6	0~0.2
0.0152	0.0060	0.0019	0.2~0.4	0.2~0.4
0.0238	0.0103	0.0011	0.6~0.8	0.2~0.4
0.0469	0.0134	0.0008	0~0.2	0.2~0.4
0.1377	0.1012	0.0194	0.2~0.4	0.4~0.6
0.6652	0.2240	0.0185	0.2~0.4	0~0.2
4.4192	6.1132	2.1115	0.2~0.4	0.6~0.8
7.2143	2.1765	0.1524	0~0.2	0~0.2

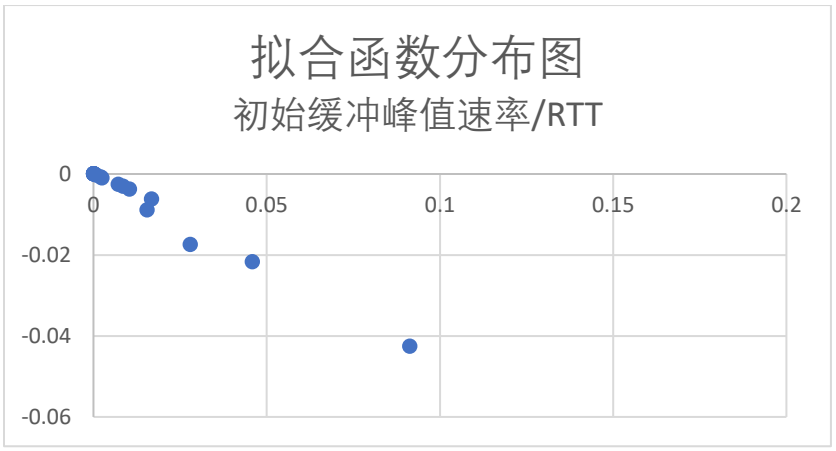
8.2 筛选函数

将拟合出来的多项式函数进行分类处理，找出大致相同的函数关系所对应的其他两个控制相同的变量的区间范围，从而得出最终的分段函数关系式。

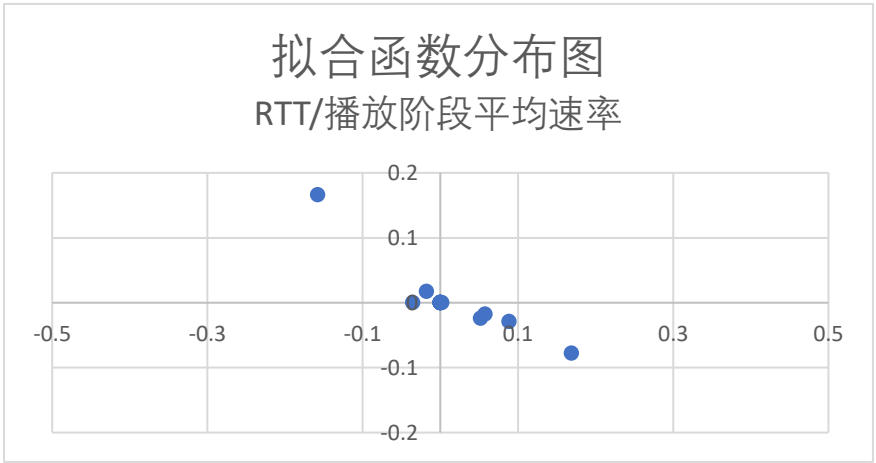
播放阶段平均下载速率与初始缓冲时延的函数关系



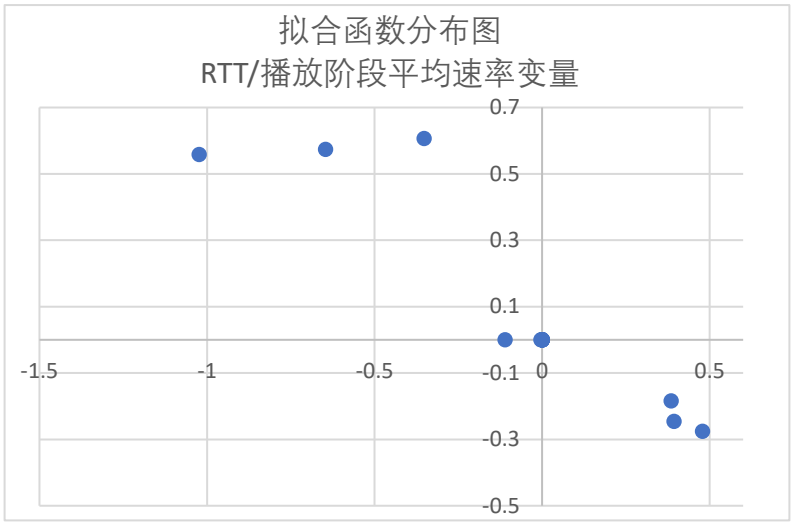
播放阶段平均下载速率与卡顿占比的函数关系



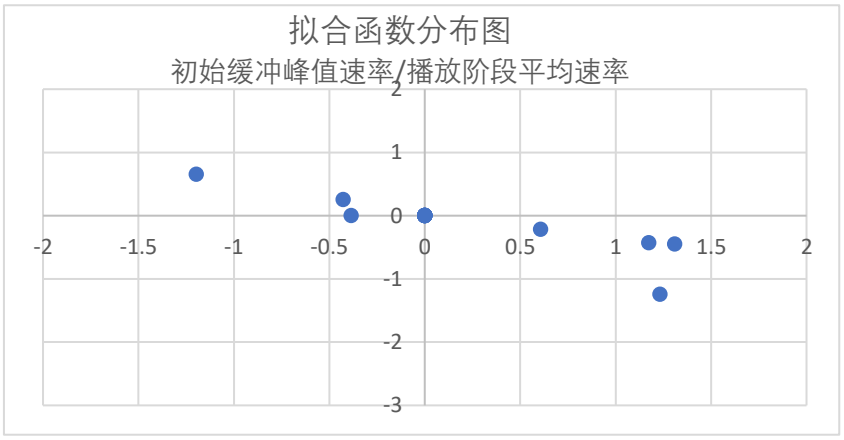
初始缓冲峰值速率与卡顿占比的函数关系



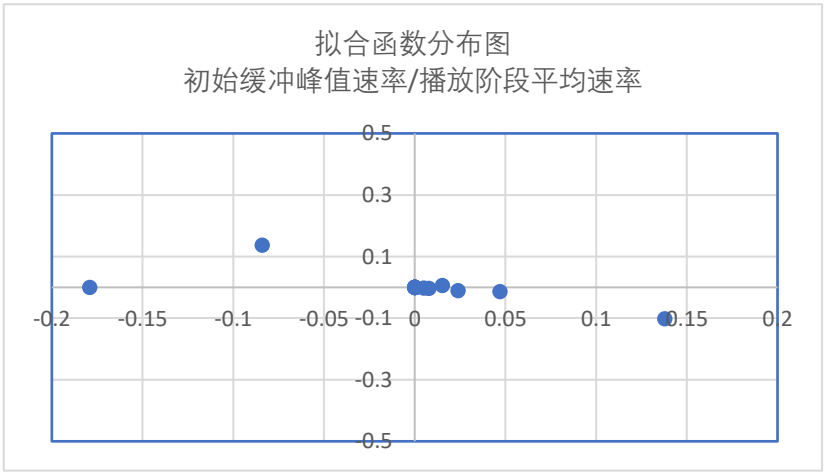
初始缓冲峰值速率与初始缓冲延时的函数关系



RTT与初始缓冲延时的函数关系



RTT与初始缓冲延时的函数关系



8.3 函数反归一化

由于我们得到的函数关系都是由归一化后的数据处理所得到的，所以我们还需按以下的公式进行反归一化：

$$\begin{cases} X_0 = \dfrac{X - X_{min}}{X_{max} - X_{min}} \\ Y_0 = \dfrac{Y - Y_{min}}{Y_{max} - Y_{min}} \\ Y_0 = aX_0^2 + bX_0 + C \end{cases}$$

$$Y = \left[\frac{a}{(X_{max} - X_{min})^2}X_0^2 + \left(\frac{b}{X_{max} - X_{min}} - \frac{2aX_{min}}{(X_{max} - X_{min})^2}\right)X_0 + \frac{aX_{min}^2}{(X_{max} - X_{min})^2} - \frac{bX_{min}}{X_{max} - X_{min}}\right]$$

$$\cdot (Y_{max} - Y_{min}) + Y_{min}$$

3.6模型结果的得出

得到的最终的分段函数关系式如下：

初始缓冲峰值速率 – 延时

$$y = \begin{cases} -0.000001x^2 + 0.1090x - 2680.63 & 11738.8 < Average < 17603.2 & 41.8 < RTT < 68.8 \\ 0x^2 - 0.0466x + 2823.83 & 5874.4 < Average < 11738 & 68.8 < RTT < 95.4 \\ 0x^2 + 0x + 506 & 11738 < Average < 29332 & 95.4 < RTT < 122.2 \end{cases}$$

初始缓冲峰值速率 – 卡顿占比

$$y = \begin{cases} 0x^2 + 0.000005x - 0.1302 & 5874.4 < Average < 11738.8 & 41.8 < RTT < 68.8 \\ 0x^2 - 0.000005x + 0.0053417 & 5874.4 < Average < 11738.8 & 68.8 < RTT < 95.4 \end{cases}$$

RTT – 延时

$$y = \begin{cases} -0.7153x^2 + 78.2878x - 1354.87 & 95656.6 < Initial < 127440 & 578.4 < Average < 11738.8 \\ 1.9630x^2 - 155.212x + 4025.50 & 32088.2 < Initial < 95656.6 & 10 < Average < 5874.4 \end{cases}$$

RTT – 卡顿占比

$$y=\begin{cases}0x^2+0x-0 & 0<Initial<32088.2 & 10<Average<29332\\ 0.000038x^2-0.000849x-0.1063 & 32088.2<Initial<63872.4 & 10<Average<17603.2\\ 0.0110x^2-2.367529x+127.3539 & 32088.2<Initial<63872.4 & 17603.2<Average<29332\\ 0x^2+0x-0 & 63872.4<Initial<159225 & 10<Average<29332\end{cases}$$

平均速率 – 延时

$$y=\begin{cases}0.000176x^2-2.201067x+10365.78 & 304<Initial<32088.2 & 41.8<RTT<68.8\\ 0.000001x^2+0.010566x+760.154 & 32088.2<Initial<127440.8 & 41.8<RTT<68.8\end{cases}$$

平均速率 – 卡顿占比

$$y=\begin{cases}0x^2-0x+0 & 63872.4<Initial<159255 & 15<RTT<149\\ 0x^2+0.000006x+0.015122 & 304<Initial<159225 & 15<RTT<95.4\end{cases}$$

8.4 模型结果的评估和改进

优点：该模型的特点在于我们采用了模糊聚类的方法，设定了截取水平 λ，对所给的矩阵进行了一系列的合成改造，生成了模糊等价矩阵，得到了样本属于各个类别的不确定性程度，表达了样本类属性的中介性，即建立起了样本对于类别的不确定性描述，更能客观的反映实际参数之间的函数关系。

缺点：我们拟合所采用的二次多项式不能很好的符合实际的函数关系，最终所得到的分段函数的适用区间也十分有限，不能很好的适用于所有样本。

本模型可以改进的地方在于改进拟合的函数类型，以此来提高拟合的精度。

参考文献

[1] HOLLAND J H. Adaptation in Natural and Artificial Systems [M]. Cambridge: MIT Press, 1992.

[2] 张袅娜, 张德江, 冯勇. 基于混沌遗传算法的柔性机械手滑模控制器 优化设计 [J]. 控制理论与应用, 2008, 25(3): 451 - 455. (ZHANG Niaona, ZHANG Dejiang, FENG Yong. The optimal design of terminal sliding controller for flexible manipulators based on chaotic genetic algorithm [J]. Control Theory & Applications, 2008, 25(3): 451 - 455.)

9. 附录

遗传算法的主要代码即 GA.cpp，请注意支撑材料的 cpp 代码均采用 UTF-8 编码，C++ 编译运行环境：编译平台: Mac OS Sierra 10.12.4

编译器：Apple LLVM version 8.1.0 (clang-802.0.42)

语言标准 C++ 11

编译选项： -O3 -std=C++11

注意：由于采用了 c++11 的语言标准，请务必使用 -std=c++11 编译选项进行编译。

```
/*
*   GA.cpp
*   Cpp Project for 2017 ZJU Mathematical Modeling Contest.
*   Team production.
*   Team member: Jiang Wen, He Kangrui, Cao Zhefeng.
*   This file was created by Wen Jiang on May 3, 2017.
*   Copyright (c) 2017 Wen Jiang. All rights reserved.
*/

#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <deque>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>
// Constant definetion.
const int IterateGeneration = 100;
const double Mutation_Probability = 0.001;
const int Row_Max = 89266;
const int Col_Max = 5;
const int Individual_Max = 20000;
const int Chromosome_Legth = 28;
const int CoeffNum = 6 * 3 + 1;
const int Depend_Var = 3;
const int Cal_Fit_Num = 800;
// const int Generation_Max = 1000;
const int HexChromLength = 8;
const double Gene_Recombination_Probability = 0.5;
// Use new feature in C++11 to initialize Gene encode mapings.
```

```
std::map<std::string, char> BinToHex = {
    {"0000", '0'}, {"0001", '1'}, {"0010", '2'}, {"0011", '3'},
    {"0100", '4'}, {"0101", '5'}, {"0110", '6'}, {"0111", '7'},
    {"1000", '8'}, {"1001", '9'}, {"1010", 'A'}, {"1011", 'B'},
    {"1100", 'C'}, {"1101", 'D'}, {"1110", 'E'}, {"1111", 'F'}};
std::map<char, std::string> HexToBin = {
    {'0', "0000"}, {'1', "0001"}, {'2', "0010"}, {'3', "0011"},
    {'4', "0100"}, {'5', "0101"}, {'6', "0110"}, {'7', "0111"},
    {'8', "1000"}, {'9', "1001"}, {'A', "1010"}, {'B', "1011"},
    {'C', "1100"}, {'D', "1101"}, {'E', "1110"}, {'F', "1111"}};
struct Individual {
  char Chrom[CoeffNum][Chromosome_Legth];  // Chromosome
  double Coeff[CoeffNum];                  // Coefficient of our funcitons.
  double Fitness;
  double Surive;
  bool Alive;
  Individual() {
    memset(Chrom, 0, sizeof(Chrom));
    memset(Coeff, 0, sizeof(Coeff));
    Fitness = 0;
    Surive = 0;
    Alive = false;
  }  // Constructo funcions.
};
// Global var:
// 2D array Data is used to read origin data.
double Data[Row_Max][Col_Max];
Individual Unit[Individual_Max];
// Struct array of individuals.
std::vector<Individual> BestUnit;
std::vector<Individual> WorstUnit;
// To memory the best and worst individuals in every generation.
int Generation = 0;
int Surive_Num = 0;
// Unit[GeneID].Coeff[0] is the constant.
// 1~6 is about x1. 7~13 is about x1
bool Compare_Ind(Individual Ind1, Individual Ind2) {
  return (Ind1.Fitness < Ind2.Fitness);
}  // Compare funciton is prepared for calling function std::sort.
char* Encode(double Code, char* Chrome);
double Decode(char* Chrome);
double Function(const double* pCoeff, double x1, double x2, double x3);
bool Load_Data();
bool Write_Individual(const Individual* pUnits_WI);
bool DEBUG_Write(const Individual* pDEBUG, int RangeInd, int Range_Coeff);
bool Write_Best();
bool Write_Fitness();
void Init_Individual(Individual* pUnits_II);
```

```
void Calculate_Fitness(Individual* pUnits_CF);
int Selction(Individual* pUnits_CF);
void Gene_Mutation(Individual* pUnit_GM_Begin, int Surive_GM);
void Gene_Recombination(Individual* pUnit_GR_Begin, int Surive_GR);
void New_Generation(Individual* pUnit_NG_Begin, int Surive_NG);
int main() {
  srand(time(NULL));
  if (!Load_Data()) {
    printf("Error in Load Data!\n");
  }
  int order, Survival_Num;
  Init_Individual(Unit);
  do {
    for (int iter = 0; iter < IterateGeneration; iter++, Generation++) {
      //Go in this loop means a new generation.
      Calculate_Fitness(Unit);
      //Calculate fitness.
      Survival_Num = Selction(Unit);
      //Count surival num.
      Gene_Mutation(Unit, Surive_Num);
      //Gene Mutation.
      Gene_Recombination(Unit, Survival_Num);
      //Gene recombination.
      New_Generation(Unit, Surive_Num);
      //Generate next generation.
      printf("After %d Generations:\n", Generation);
      printf("  The Best perfom indvidual's fitness is %.6f\n",
            (BestUnit.back()).Fitness);
      printf("  The worst perfom indvidual's fitness is %.6f\n",
            (WorstUnit.back()).Fitness);
    }
    printf(
        "After %d Generations:\n type the non-zero number to continue "
        "iterations, type zero to end the program.\n",
        Generation);
  } while (scanf("%d", &order) && order);
  Write_Best();//Write best unit.
  Write_Fitness();//write fitness statistics.
  return 0;
}
char* Encode(double Code, char* Chrome) {
  char HexBuf[8 + 1];  // Character array for buffer.
  if (Code > 10 || Code < -10) Code = fmod(Code, 10);
  sprintf(HexBuf, "%07X", (int)floor(Code * 1000000));
  // Transport to hex intergers.
  for (int Iter_HB = 0; Iter_HB < HexChromLength; Iter_HB++) {
    sprintf(Chrome + 4 * Iter_HB, "%s", HexToBin[HexBuf[Iter_HB]].c_str());
  }  // Trans for by the order of hex intergers.
```

```
  return Chrome;
}
double Decode(char* Chrome) {
  char DecodeBuf[8 + 1];
  memset(DecodeBuf, 0, sizeof(DecodeBuf));
  // Initialize.
  std::string Decoded = Chrome;
  for (int Iter_BH = 0; Iter_BH < HexChromLength; Iter_BH++) {
    DecodeBuf[Iter_BH] = BinToHex[Decoded.substr(Iter_BH * 4, 4)];
    // Get 4 bits every to and use the mapping to find whcich hex number it
    // representing.
  }
  int decodedNum;
  sscanf(DecodeBuf, "%X", &decodedNum);
  // Transfor hex string to hex number.
  return (decodedNum * 1.0 / 1000000);
}
double Function(const double* pCoeff, double x1, double x2, double x3) {
  return (pCoeff[0] + pCoeff[1] * exp(pCoeff[2] * x1) +
          pCoeff[3] * log(1 + x1) + pCoeff[4] * x1 + pCoeff[5] * x1 * x1 +
          pCoeff[6] * (1.0 / (1 + x1)) + pCoeff[7] * exp(pCoeff[8] * x2) +
          pCoeff[9] * log(1 + x2) + pCoeff[10] * x2 + pCoeff[11] * x2 * x2 +
          pCoeff[12] * (1.0 / (1 + x2)) + pCoeff[13] * exp(pCoeff[14] * x3) +
          pCoeff[15] * log(1 + x3) + pCoeff[16] * x3 + pCoeff[17] * x3 * x3 +
          pCoeff[18] * (1.0 / (1 + x3)));
  // The target function set by us previously.
}
bool Load_Data() {
  FILE* fpData = NULL;
  fpData = fopen("./Data.txt", "r");
  if (fpData == NULL) return false;
  for (int i = 0; i < Row_Max; i++) {
    for (int j = 0; j < Col_Max; j++) {
      fscanf(fpData, "%lf", &Data[i][j]);
    }
  }
  if (fclose(fpData) != 0) return false;
  return true;
}
bool Write_Individual(const Individual* pUnits_WI) {
  FILE* fpCoeff = NULL;
  fpCoeff = fopen("./Coeff.txt", "a+");
  if (fpCoeff == NULL) return false;
  fprintf(fpCoeff, "Affter %d Generations:\n", Generation);
  for (int GeneID_WI = 0; GeneID_WI < Individual_Max;
       GeneID_WI++, pUnits_WI++) {
    fprintf(fpCoeff, "No.%3d Fitness:%f\n", GeneID_WI, pUnits_WI->Fitness);
    for (int iWI = 0; iWI < CoeffNum; iWI++) {
```

```
      fprintf(fpCoeff, "%.6f ", pUnits_WI->Coeff[iWI]);
    }
    fprintf(fpCoeff, "\n");
  }
  if (fclose(fpCoeff) != 0) return false;
  return true;
}
bool DEBUG_Write(const Individual* pDEBUG, int RangeInd, int Range_Coeff) {
  FILE* fpDEBUG = NULL;
  fpDEBUG = fopen("./DEBUG.txt", "w");
  if (fpDEBUG == NULL) return false;
  for (int GeneID_WI = 0; GeneID_WI < RangeInd; GeneID_WI++, pDEBUG++) {
    fprintf(fpDEBUG, "No.%3d Fitness:%f\n", GeneID_WI, pDEBUG->Fitness);
    for (int iWI = 0; iWI < Range_Coeff; iWI++) {
      fprintf(fpDEBUG, "%.6f ", pDEBUG->Coeff[iWI]);
    }
    fprintf(fpDEBUG, "\n");
  }
  if (fclose(fpDEBUG) != 0) return false;
  return true;
}
bool Write_Best() {
  FILE* fpBest = NULL;
  fpBest = fopen("./Best.txt", "a+");
  if (fpBest == NULL) return false;
  int cnt_WB = 0;
  for (std::vector<Individual>::iterator it_WB = BestUnit.begin();
       it_WB != BestUnit.end(); ++it_WB, ++cnt_WB) {
    fprintf(fpBest, "Generation %d Fitness:%.6f\n", cnt_WB, it_WB->Fitness);
    for (int iWB = 0; iWB < CoeffNum; iWB++) {
      fprintf(fpBest, "%.6f ", it_WB->Coeff[iWB]);
    }
    fprintf(fpBest, "\n");
  }
  if (fclose(fpBest) != 0) return false;
  return true;
}
bool Write_Fitness() {
  FILE *fpBFitness = NULL, *fpWFitness = NULL;
  fpBFitness = fopen("./Best_Fitness.txt", "a+");
  fpWFitness = fopen("./Worst_Fitness.txt", "a+");
  if (fpBFitness == NULL || fpWFitness == NULL) return false;
  for (std::vector<Individual>::iterator it_WF_Best = BestUnit.begin();
       it_WF_Best != BestUnit.end(); ++it_WF_Best) {
    fprintf(fpBFitness, "%.6f\n", it_WF_Best->Fitness);
  }
  for (std::vector<Individual>::iterator it_WF_Worst = WorstUnit.begin();
       it_WF_Worst != WorstUnit.end(); ++it_WF_Worst) {
```

```
    fprintf(fpWFitness, "%.6f\n", it_WF_Worst->Fitness);
  }
  if (fclose(fpBFitness) != 0 || fclose(fpWFitness) != 0) return false;
  return true;
}
void Init_Individual(Individual* pUnits_II) {
  for (int GeneID_II = 0; GeneID_II < Individual_Max;
       GeneID_II++, pUnits_II++) {
    for (int iII = 0; iII < CoeffNum; iII++) {
      pUnits_II->Coeff[iII] = (4.0 * rand() / RAND_MAX) - 2;
    }
  }
}
void Calculate_Fitness(Individual* pUnits_CF) {
  double(*pCheck)[4] = (double(*)[4])malloc(sizeof(double) * Cal_Fit_Num * 4);

  for (int Row_CF = 0, Rand_CF = 0; Row_CF < Cal_Fit_Num; Row_CF++) {
    Rand_CF = (int)floor((1.0 * Row_Max * rand()) / RAND_MAX);
    if (Rand_CF == Row_Max) Rand_CF = Row_Max - 1;
    pCheck[Row_CF][0] = Data[Rand_CF][0];
    pCheck[Row_CF][1] = Data[Rand_CF][1];
    pCheck[Row_CF][2] = Data[Rand_CF][2];
    pCheck[Row_CF][3] = Data[Rand_CF][Depend_Var];
  }
  for (int GeneID_CF = 0; GeneID_CF < Individual_Max;
       GeneID_CF++, pUnits_CF++) {
    pUnits_CF->Fitness = 0;
    for (int i_CF = 0; i_CF < Cal_Fit_Num; i_CF++) {
      pUnits_CF->Fitness +=
          std::abs((Function(pUnits_CF->Coeff, pCheck[i_CF][0], pCheck[i_CF][1],
                             pCheck[i_CF][2])) -
                   pCheck[i_CF][3]);
    }
  }
  free(pCheck);
}
int Selction(Individual* pUnits_CF) {
  // Nature selction

  Individual* Temp_ind_begin =
      (Individual*)malloc(sizeof(Individual) * Individual_Max);
  Individual* Sel_ind_Begin =
      (Individual*)malloc(sizeof(Individual) * Individual_Max);
  Individual* Sel_ind = Sel_ind_Begin;
  Individual* pIter = pUnits_CF;
  Individual* Temp_ind = Temp_ind_begin;
  for (int Cp_Id = 0; Cp_Id < Individual_Max; Cp_Id++, Temp_ind++, pIter++) {
    *Temp_ind = *pIter;
```

```
  }
  std::sort(Temp_ind_begin, Temp_ind_begin + Individual_Max - 1, Compare_Ind);
  // Write_Individual(Temp_ind_begin);
  // DEBUG_Write(Temp_ind_begin, Individual_Max, CoeffNum);
  BestUnit.push_back(*Temp_ind_begin);
  WorstUnit.push_back(*(Temp_ind_begin + Individual_Max - 1));
  Surive_Num = 0;
  Temp_ind = Temp_ind_begin;
  for (int Sel_Id = 0; Sel_Id < Individual_Max; Sel_Id++, Temp_ind++) {
    Temp_ind->Surive = 1.0 - (1.0 / Individual_Max) * Sel_Id;
    if ((((double)rand()) / RAND_MAX) <= Temp_ind->Surive) {
      Surive_Num++;
      Temp_ind->Alive = true;
      *Sel_ind = *Temp_ind;
      Sel_ind++;
    } else {
      Temp_ind->Alive = false;
    }
  }
  pIter = pUnits_CF;
  Individual* Sel_Iter = Sel_ind_Begin;
  for (int Cp_Id = 0; Cp_Id < Surive_Num; Cp_Id++, pIter++, Sel_Iter++) {
    *pIter = *Sel_Iter;
  }
  // DEBUG_Write(Sel_ind_Begin);
  free(Temp_ind_begin);
  free(Sel_ind_Begin);
  return Surive_Num;
}
void Gene_Mutation(Individual* pUnit_GM_Begin, int Surive_GM) {
  Individual* pUnit_GM = pUnit_GM_Begin;
  for (int it_GM = 0; it_GM < Surive_GM; it_GM++, pUnit_GM++) {
    for (int Coeff_GM = 0; Coeff_GM < CoeffNum; Coeff_GM++) {
      // Encode
      Encode(pUnit_GM->Coeff[Coeff_GM], (pUnit_GM->Chrom)[Coeff_GM]);
      // Mutation
      if (((double)rand()) / RAND_MAX < Mutation_Probability) {
        int Mutation_Index = (int)(27.0 * rand() / RAND_MAX);
        (pUnit_GM->Chrom)[Coeff_GM][Mutation_Index] =
            (pUnit_GM->Chrom)[Coeff_GM][Mutation_Index] == '0' ? '1' : '0';
      }
      // decode
      pUnit_GM->Coeff[Coeff_GM] = Decode((pUnit_GM->Chrom)[Coeff_GM]);
    }
  }
}
void Gene_Recombination(Individual* pUnit_GR_Begin, int Surive_GR) {
  Individual* pUnit_GR = pUnit_GR_Begin;
```

```
  for (int it_GR = 0; it_GR < Surive_GR; it_GR++, pUnit_GR++) {
    for (int Coeff_GR = 0; Coeff_GR < CoeffNum; Coeff_GR++) {
      // Recombination.
      if (((double)rand()) / RAND_MAX < Gene_Recombination_Probability) {
        int Recombination_Index = 1.0 * (Surive_GR - 1) * rand() / RAND_MAX;
        double Temp_Coeff = (pUnit_GR->Coeff)[Coeff_GR];
        (pUnit_GR->Coeff)[Coeff_GR] =
            (pUnit_GR_Begin[Recombination_Index]).Coeff[Coeff_GR];
        (pUnit_GR_Begin[Recombination_Index]).Coeff[Coeff_GR] = Temp_Coeff;
      }
    }
  }
}
void New_Generation(Individual* pUnit_NG_Begin, int Surive_NG) {
  int Half = Individual_Max / 2;
  if (Surive_NG < Half) {
    for (int iter_NG = Surive_NG, i_Better = 0; iter_NG < Half;
         iter_NG++, i_Better++) {
      pUnit_NG_Begin[iter_NG] = pUnit_NG_Begin[i_Better];
    }
  }
  for (int iterNG = 0; iterNG < Half; iterNG++) {
    pUnit_NG_Begin[Half + iterNG] = pUnit_NG_Begin[iterNG];
  }
}
```