基于极大似然估计的模拟事件信号的分辨与拟合

余荫铠，王誉晨

中山大学 物理学院，广州 510275

摘　要：粒子物理实验的数据处理过程常常要求从混杂的不同已知事件信号中分辨各个事件发生的次数和比例，对数似然函数分析是解决这个问题的有效方法。我们提出了一种更为高效的快速对数似然函数分析，同时将对数似然函数分析推广到了更高维度，得以利用粒子物理实验中得到的多维数据，尤其是具有相关性的多维数据来提高事件分辨的准确程度。

关键词：极大似然估计，粒子物理实验数据处理，二维正态分布，数值稳定性

1　引言

粒子物理实验会产生大量的实验数据，这些数据需要通过分析来获得有用的信息。在一次物理事件中，探测器测得的数据会包含大量的不相关的测量量，构成了这次事件的信息。通常，数据分析最后归结于如何通过仅包含事件“信号”和“背景”的简单样本，将拥有大量数据的事件划分到很少的分类中。

粒子物理实验数据分析的一种常见的范式是：已知可能发生的事件的类型，需要从混杂的事例集合中分辨发生了哪些事件、以及它们发生的次数和比例。我们的工作主要围绕这一问题来具体进行。

极大似然估计是数理统计中的参数估计的基本方法之一。直接使用极大似然估计求解事件发生的次数，可能会在置信区间的求解过程中出现数值病态的问题，因此中山大学基础物理实验教学团队在他们的实验讲义 [1] 中给出了基于分区负对数似然函数的对数似然分析（LLF）方法，我们将在2中对其进行概述和重复。我们注意到该方法在数值计算的过程中存在被浪费的约束条件，从这里入手我们提出了快速对数似然分析方法（FLLF），极大提高了运算效率，这一工作在3中展示。

如何将 LLF 扩展到更高维度，是同学们的研究热点之一。莫梁虹 [2] 和罗俊平 [3] 分别提出了仍然基于一维 LLF 的降维方法，以应对更高维的数据。不同于他们的思路，我们在4提出了多维 LLF 方法，充分利用了不同维度信号的相关性，给出了偏差更小方差也更小的拟合结果，提高了 LLF 的事件分辨能力。

针对粒子物理实验数据处理的其他范式，最后在5我们还对 LLF 的多自由度拟合结果进行了讨论。

2　对数似然函数分析

在这一部分，我们简要介绍分区极大似然估计的原理，并以从能谱中分辨两个混杂的已知事件的比例为例，说明分区对数似然函数分析的具体方法和效果。这些工作都基于中山大学基础物理实验团队提供的讲义 [1]。

2.1　极大似然估计

在粒子物理实验中，我们的实验装置往往会检测到大量的事例。我们统计这些事例的某一观测值，比如能量值，对其能量范围进行分区作出统计直方图，可以得到离散化能谱。在物理上，对于已知的一个或多个事件，我们可以根据其能量分布函数计算出各个区间统计到的事例数的期望值。为方便描述原理和算法呈现，我们假设某一事件 j

的能量分布服从（归一化）正态分布

$$f_j\left(x;\mu_j,\sigma_j\right)=\frac{1}{\sigma_j\sqrt{2\pi}}\mathrm{e}^{-\frac{\left(x-\mu_j\right)^2}{2\sigma_j^2}} \tag{1}$$

那么分布在第 i 个区间的事例数的期望值为

$$\lambda_i\left(\widetilde{N},\tilde{\mu},\tilde{\sigma}\right)=\sum_j N_j\int_i f_j\left(x;\mu_j,\sigma_j\right)\mathrm{d}x \tag{2}$$

其中 N_j 为事件 j 发生的次数，简记 $\widetilde{N}=\{N_j\}$，$\tilde{\mu}=\{\mu_j\}$，$\tilde{\sigma}=\{\sigma_j\}$。对第 i 个区间的积分 $\int_i$ 在实际计算的过程中可以使用大量（数量级远大于待分析事例数）的模拟撒点统计来估算，也可以直接积分。

在待分析事例数不太大的情况下，落入第 i 个区间的事例数服从泊松分布，则对于某一观测结果，其似然函数（即发生的概率）为

$$L\left(\tilde{k};\tilde{\lambda}\left(\widetilde{N},\tilde{\mu},\tilde{\sigma}\right)\right)=\prod_i\frac{\lambda_i^{k_i}\mathrm{e}^{-\lambda_i}}{k_i!} \tag{3}$$

其中 $\tilde{k}=\{k_i\}$ 表示实际观测到各个区间的事例数，$\tilde{\lambda}=\{\lambda_i\}$ 为各个区间的事例数期望值。

根据相关统计学知识，最可能的事件参数值使得似然函数取得最大值。对于特定的观测结果，为分析数据来“拟合”出待求的事件参数，我们以 $\tilde{k}$ 为参数，把 L 看作 $\widetilde{N},\tilde{\mu},\tilde{\sigma}$ 的函数，得到一个多元函数求极大值点的优化问题。在实际的数据处理中，为了提高数值计算的效率和数值稳定性，通常将这个问题等效为求似然函数的负对数的极小值点

$$l\equiv-\ln L=\sum_i\left(\lambda_i-k_i\ln\lambda_i\right) \tag{4}$$

对于特定的观测结果，我们忽略了常数项项 $\ln(k_i!)$，它不表现对 $\widetilde{N},\tilde{\mu},\tilde{\sigma}$ 的响应。由此 $\widetilde{N},\tilde{\mu},\tilde{\sigma}$ 的极大似然估计值 $\widetilde{N}^*,\tilde{\mu}^*,\tilde{\sigma}^*$ 满足

$$\min l=l(\widetilde{N}^*,\tilde{\mu}^*,\tilde{\sigma}^*) \tag{5}$$

利用 χ^2 分布和泊松分布相似的特点，根据 Wilks 定理 [4] 给出

$$\chi^2=-2(l'(\theta)-l_{\min}) \tag{6}$$

其中 $l_{\min}$ 为似然函数的负对数的最小值，$l'(\theta)$ 为似然函数的负对数的最小值点附近改变某个参数 $\theta\in\widetilde{N},\tilde{\mu},\tilde{\sigma}$ 所对应的似然函数值的负对数。在最小值点附近扫描 θ，当 $\chi^2=1$ 时，得到置信水平为 1σ 的 θ 上界 θ^+ 和下界 θ^-，置信区间为 $[\theta^-,\theta^+]$。

2.2 基于极大似然估计的事件分辨

中山大学基础物理实验团队提供的讲义 [1] 提供了基于分区对数似然函数分析的一个案例，用于分辨两个已知事件的混杂比例。此时，$\mu_A,\mu_B,\sigma_A,\sigma_B$ 都为已知的确定值，待求的事件参数为 N_A,N_B。该问题具体化为二元负对数似然函数求最小值点的问题，(5) 具体为

$$\min l=l(N_A^*,N_B^*) \tag{7}$$

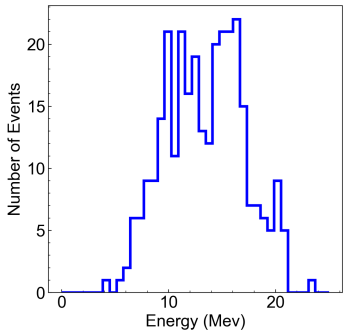


图 1 模拟赝事件的事例数分布直方图

在下面的基于 python 的数值计算中，用参数值 $\mu_A=10$，$\mu_B=15$，$\sigma_A=2$，$\sigma_B=3$，$N_A=100$，$N_B=200$ 生成共 300 个正态分布的模拟赝事例，分区数量为 $n=40$ 作出其直方图。(2) 中的积分即分区概率密度使用蒙特卡罗方法撒 10000 个正态分布点并归一化来估算

$$\int_i f_j\left(x;\mu_j,\sigma_j\right)\mathrm{d}x\approx\mathrm{MC}_j(i) \tag{8}$$

其中 $\mathrm{MC}_j(i)$ 表示第 j 个事件在第 i 个区间上的归一化蒙特卡罗撒点数。最终得到似然函数的具体表达式

$$l(N_A,N_B)=\sum_{i=1}^{n}\left(\lambda_i-k_i\ln\lambda_i\right) \tag{9}$$

其中

$$\lambda_i=\mathrm{MC}_A(i)N_A+\mathrm{MC}_B(i)N_B \tag{10}$$

考虑真值 $N_A=100$，$N_B=200$ 附近的参数空间，作出 $l(N_A,N_B)$ 的等高线图2，可以直观地看到 $l(N_A,N_B)$ 有一个稳定的极小值点在真值点附

近——该方法是数值稳定的。

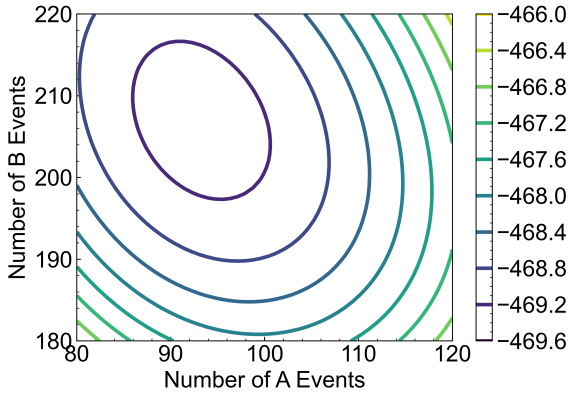


图 2 N_A, N_B 参数空间的负对数似然函数等高线

在图2所示的极小值点附近取迭代初值，使用 Nelder-Mead 方法 [5]，最小化负对数似然函数 $l(N_A, N_B)$，拟合出的极大似然点为

$$\begin{aligned} N_A^* &= 93.15^{+12.76}_{-12.03} \\ N_B^* &= 206.85^{+16.74}_{-15.96} \end{aligned} \tag{11}$$

其中置信水平为 1σ 的上下置信区间由 (6) 取 $\chi^2 = 1$ 的条件确定，详细的计算过程参考附录代码C.1。最后，据此作出拟合的事例数分布直方图3。

由 (11) 我们可以看出生成数据所用的真值落在置信水平为 1σ 内，这说明分区负对数似然函数分析可以有效地分辨混杂的 A, B 两类事件。此外，图3中大部分分区上的拟合值也都落在模拟赝事例数的误差范围内，给出了较好的拟合结果。

值得一提的是，在这个案例中，对数似然函数分析相较于直接的似然函数分析的数值稳定性提升得以体现。在附录C.1中我们也展示了使用对数似然函数分析的具体过程。使用对数似然函数对这个案例进行分析，仍然可以拟合出极大似然点，一致于 (11) 所给的拟合中心值，而其数值的病态则体现在置信区间的求解中。事实上，采用对数似然函数分析，得到的 $l(N_A, N_B)$ 极小值为

$$\min l = l(N_A^*, N_B^*) = -469.38 \tag{12}$$

而采用直接似然函数分析得到的 $L(N_A, N_B)$ 极大值为

$$\max L = L(N_A^*, N_B^*) = 8.3728 \times 10^{-32} \tag{13}$$

后者处于相当小的数量级——而在求解置信区间时，我们要求

$$\chi^2 = -2(l'(\theta) - l_{\min}) = -2 \ln \frac{L'(\theta)}{L_{\max}} = 1 \tag{14}$$

此时对于直接似然函数分析的结果，需要计算 $L'(\theta)$ 和 $L_{\max}$ 两个很小的数的比值，这是一个典型的数值病态问题。即使这时再将 $\ln \frac{L'(\theta)}{L_{\max}}$ 化为两个对数值的差来计算，仍不能避免其病态性，因为对数函数对很小的自变量的误差是相当敏感的。

此外，对于中大教学团队的这个典型的模型，我们在附录B中讨论 A, B 两类事件的比例及总量以评估该模型的分辨能力范围，这只是调节参数的问题，就不占用正文的篇幅了。唯一值得交代的是，与人们通常的直觉不同，该算法的拟合效果不敏感与直方图的光滑程度——它取决于分区数量。我们在相邻数量级改变分区数量，并重新求解模型，发现 N_A, N_B 的相对置信容差 (图4) 几乎不随分区数量的变化有可观的改变，仅是在分区过密时不稳定。

3 快速对数似然函数分析

前文我们展示了中大教学团队提供的对数似然函数分析方法。接下来我们将基于已有的讨论，展示我们对这个方法的优化和拓展。快速对数似然函数分析（FLLF）相较于对数似然函数分析（LLF），在运算效率方面有很大的提升，这在实际的粒子物理实验处理中是相当重要的。

3.1 被浪费的约束条件

在实际的粒子物理实验中，所探测到是事例总数 N_{total} 往往是已知的。那么事件 A, B 所发生的次数 N_A, N_B 应当满足约束

$$N_A + N_B = N_{\text{total}} \tag{15}$$

如果不考虑噪声和杂质（比如前文的仅有 A, B 两个赝事件的模型），(15) 严格成立，原二元函数的极值问题实际上是一元函数的极值问题。

参数空间（图2）冗余了一个自由度，等高线的变化在直线 $N_A = N_B$ 方向上展现出平凡的趋势：极值点自然是出现在 $N_A + N_B$ 不能太大也不会太小处，该自由度是必定收敛的。尽管二维的最优值点很可能不落在 (15) 对应的直线上，但是这个偏差并不对应于物理的更优拟合结果。甚至——我们观察图2，可以看出等高线为椭圆形，在

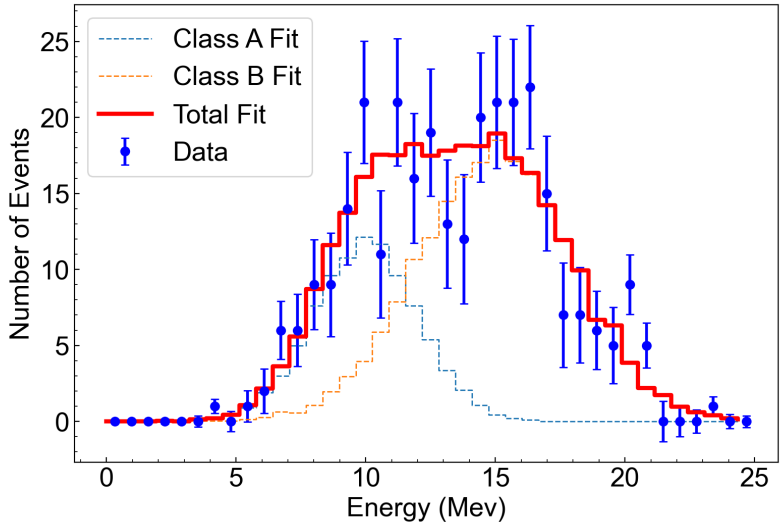


图 3 事例数分布直方图拟合结果

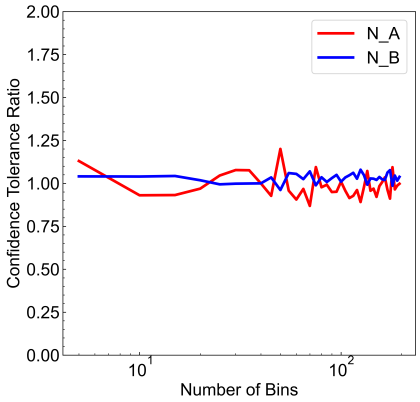


图 4 N_A, N_B 的相对置信容差随分区数量的变化
相对置信容差定义为参数的置信区间长度与参数拟合中心值的比值，这里呈现的为归一化之后的结果，即纵轴的单位为分区数为 40 时的相对置信容差。

置信水平 1σ 的边界对应的等高线也是一个参数空间中的椭圆，我们在计算 N_A, N_B 的置信区间时，是将这个椭圆投影到 N_A, N_B 轴上得到一线段，事实上 N_A, N_B 的置信区间的交集为置信区间边界的外切矩形，它将置信区间放大了，损失了图2中更精细的数据信息；而如果我们将求解范围约束在直线 (15) 上，该直线和置信区间边界椭圆的交集为一斜线段，其在 N_A, N_B 上的投影范围必定小于等于椭圆的外接矩形的投影范围，从而在降维呈现的过程中完全保留了有物理意义的似然函数的置信空间信息。

我们总结添加约束 (15) 给这个拟合过程的带来的影响：

1. 求解可行域降维，大大提高求解效率和稳定性，不容易收敛到非物理的局部最优点。
2. 引入待分析数据的未利用信息，降低拟合结果与真值的偏差。
3. 排除了冗余拟合自由度，在置信空间降维为置信区间的过程中完全保留了有物理意义的的拟合信息，降低了拟合结果的方差，提高拟合精度。

在实际的物理实验中，约束 (15) 并不一定严格成立，这是因为我们无法排除是否会有除事件 A, B 之外的其他事例被接收。这是需要将约束 (15) 进行一定程度的“软化”，具体的软化形式取决于具体的实验内容以及对可能的干扰事件的推测，但

无论如何，以上三点影响，它仍然具备

3.2 加速效果

快速对数似然函数分析（FLLF），其实就是考虑约束 (15) 的极大似然函数分析（LLF），将二元函数极值问题转化为了一元函数极值问题。

为了对比 FLLF 相较于 LLF 的效率提高，我们定义增效比

$$\eta = \frac{\text{nfev(LLF)}}{\text{nfev(FLLF)}} \tag{16}$$

其中 nfev(LLF), nfev(FLLF) 分别表示 LLF 迭代次数和 FLLF 迭代次数。对于不同的求极小值算法，我们分别统计了增效比，如表1所示。

表1的结果显示 FLLF 的计算效率相较于 LLF 有较大提高，特别是对于网格搜索算法，降低一个搜索维度带来的效率增益是不可估量的。对于我们这个玩具模型而言，无论是 FLLF 还是 LLF，迭代次数都比较少，该效率增益不能直观地体现出来，而在实际的粒子物理的大样本容量的实验中，如此的效率增益是具有很大意义的。

4 多维对数似然函数分析

在2中，中大教学团队提供的模型只分析了一维信号，即不同事例的能量。在实际的粒子物理实验中，还有很多其他的信息可以同时被探测，例如速率。从而，额外的信息通常有利于事件的分辨，在拟合自由度仍为 N_A, N_B 二自由度的情况下，事例数的分布维度由能量一维变成了能量、速率二维。我们以这种情况为例展开分析，该部分的结果具有较大的可拓展性。

4.1 含相关关系的多维分类问题

高维数据分类问题是计算物理领域的经典问题。但是我们需要明确，传统的高维数据分类问题，往往是分析多个相互独立的维度，数据的不同维度的信息之间是没有相关性的。机器学习领域传统的高维数据分类算法比如有标记训练的支持向量机、无标记训练的 k 均值算法，它们都擅长处理各维度独立的高维数据。这是因为它们本质上是在高维空间中计算两类数据点的高维间隔距离，这是基于两类数据在每个维度的特征都有一定的间距，在高维空间中综合考虑每一个维度的间距，则数据在高维空间中表现出更明显的特征间隔。这些算法是不擅长处理含相关关系的多维分类问题的，因为如果维度之间有较强的相关关系，高维空间几乎不会放大数据的特征间隔。这相当于含相关关系的多维数据没有给这些算法输入更多新的有助于分类的信息。

现在我们考虑的能量、速率二维分类问题，就是含相关关系的多维分类问题。如果两个事件的能量是比较接近的，那么根据物理上的动力学关系，两个事件的也有可能是比较接近的。支持向量机、k 均值算法便不擅长处理这类问题。

极大似然估计方法是基于物理规律构造优化成本（即负对数似然函数）的，可以利用两个维度的相关关系提高事件的分辨能力，即使两个事件的能量和速度都很接近，该方法依然有很好的分辨效果。

4.2 基于相关性的多维分辨能力

相对论动力学方程给出能量和速率之间的关系

$$v = \sqrt{1 - \left(\frac{m_0 c^2}{E}\right)^2}\, c \tag{17}$$

其中 m_0 为静质量。如果不考虑涨落，只要两类粒子的静质量是有区别的（那必然），那么我们同时知道了速率和能量，就可以唯一确定该粒子的类型。

考虑粒子物理实验中的涨落，由 (17) 依然有

$$\sigma_v = \frac{\left(\frac{m_0 c^2}{E}\right)^2}{\sqrt{1 - \left(\frac{m_0 c^2}{E}\right)^2}} \frac{\sigma_E}{E} c \tag{18}$$

即速度的涨落线性相关于能量的涨落，如2中考虑能量服从正态分布，则该情形能量、速度的联合概率分布服从二维正态分布

$$\begin{aligned} f(E, v) = & \frac{1}{2\pi \sigma_E \sigma_v \sqrt{1-\rho^2}} \\ & \exp\left\{ -\frac{1}{2(1-\rho^2)} \left[\frac{(E-\mu_E)^2}{\sigma_E^2} \right.\right. \\ & \left.\left. -2\rho \frac{(E-\mu_E)(v-\mu_v)}{\sigma_E \sigma_v} + \frac{(v-\mu_v)^2}{\sigma_v^2} \right] \right\} \end{aligned} \tag{19}$$

其中 $\rho = 1$。且 μ_E, μ_v 服从 (17) 的关系。但实际的实验中，速率本身也存在涨落，不一定完全线性相关于能量的涨落，故 ρ 会略小于 1。

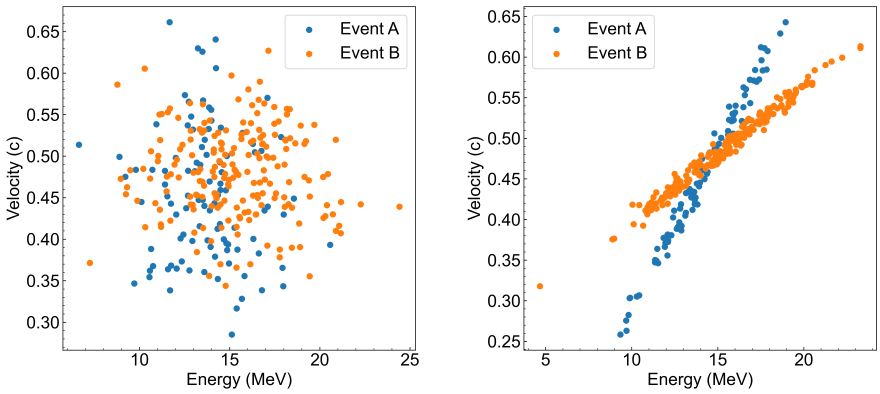
如何利用具有相关关系的多维数据提高相关关系呢? 为了便于比较，在图5中我们按照表2所示的参数值生成赝事例，在速率-能量参数空间中展

表 1 FLLF 相比于 LLF 的增效比

最小化算法	LLF 迭代次数	FLLF 迭代次数	增效比
BFGS	54	16	337%
Powell	75	22	340%
CG	120	84	142%
Nelder-Mead	93	44	211%
COBYLA	113	80	141%
L-BFGS-B	30	14	214%
TNC	207	88	235%
SLSQP	19	4	475%
trust-constr	27	10	270%
GridSearchCV	arbitrary for 2d	arbitrary for 1d	∞

表 2 二维正态分布模拟赝事例的生成参数

μ_E (MeV)	σ_E (MeV)	μ_v (c)	σ_v (c)	事例数
14	2	0.45	0.08	100
15	3	0.48	0.05	200



(a) $\rho = 0$ (b) $\rho = 0.99$

图 5 维度的相关关系对模拟赝事例分布的影响

示其分布。从图5中我们可以看到，两个维度间良好的相关性大大减少了两类事件散点的交叠区域面积。

分区对数似然函数法对这个二维参数空间进行统计，作出直方图，并结合每个区间内的实测事例数和期望事例数计算出似然函数值，这一过程是对非交叠区域占比敏感的，而几乎无关于表2中 A, B 两类事件十分接近的 μ 和十分大的 σ。对于其他的算法，比如支持向量机或者 k 均值算法，都是敏感于 μ, σ 的。

并且这一优越性只有在多维分区对数似然函数分析才得以体现，一维分区方法仍会得到较大的交叠面积，A, B 两类事件是难以区分的。在下面的实验中我们将比较它们的结果，我们所用的参数值依然如表2，并如图5(b) 一样取 $\rho = 0.99$。

如果我们采取对二维参数空间中的事例分布作一维投影，分别投影到速度和能量轴 [2]，显然都会获得非常大的交叠区域，拟合结果的置信区间就会非常大，我们采用莫梁虹的文章中的方法，根据表2的参数拟合得到结果为
向能量轴投影，并作一维对数似然函数分析：

$$\begin{aligned} N_A^* &= 69.07^{+30.79}_{-31.33} \\ N_B^* &= 230.93^{+34.67}_{-32.44} \end{aligned} \tag{20}$$

向速率轴投影，并作一维对数似然函数分析：

$$\begin{aligned} N_A^* &= 118.70^{+32.88}_{-30.40} \\ N_B^* &= 181.30^{+32.40}_{-32.88} \end{aligned} \tag{21}$$

可见拟合结果的置信区间都非常大，拟合结果并不精准了。而莫梁虹在 [2] 中提出的方法为将以上两个结果的置信区间取交集，则是没有正确考虑两种投影结果的拟合中心值的差异了。

下面我们使用多维对数似然函数分析。该方法相比于前文中大教学团队的对数似然函数分析2，差别仅仅在于把 (8) 改写为二维形式：

$$\iint_i f_j\left(E, v; \mu_{Ej}, \mu_{vj}, \sigma_{Ej}, \sigma_{vj}\right) \mathrm{d}E\mathrm{d}v \approx \mathrm{MC}_j(i) \tag{22}$$

这里的蒙特卡罗撒点服从相应的二维正态分布即可。

最后我们拟合得到的结果为

$$\begin{aligned} N_A^* &= 105.76^{+12.55}_{-11.73} \\ N_B^* &= 195.85^{+15.82}_{-15.13} \end{aligned} \tag{23}$$

将这个结果在图7中与莫梁虹使用一维方法求解的结果对比，显然可以看出我们所采用的二维对数似然法给出的结果具有更小的偏差，置信区间也给得更窄，事例分辨的准确度明显提升。

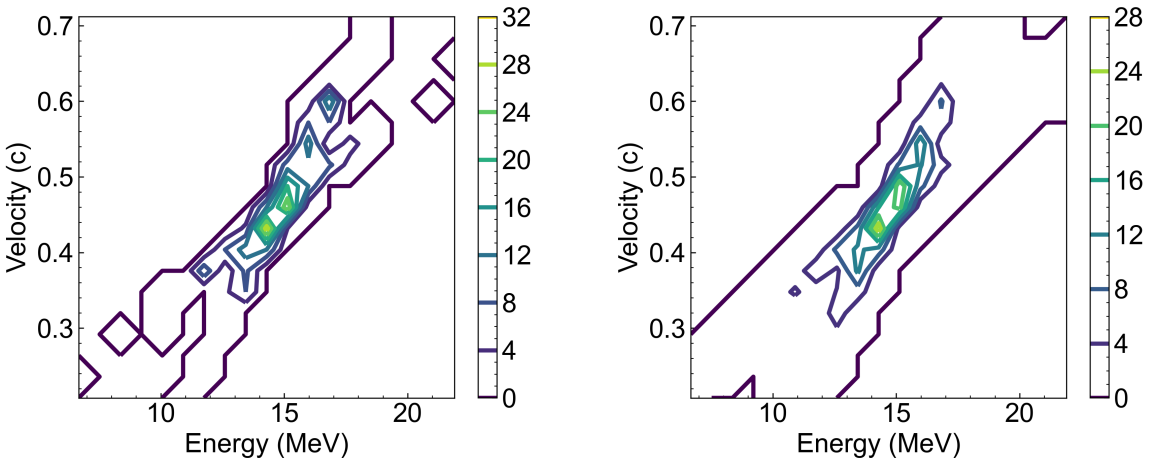
同时，也正如我们前面的图3一样，我们将求解得到的参数 (23) 代回 (2)，拟合出速率-能量参数空间直方图，并将其与原模拟事例的直方图对比，如图6所示。从图6可以看出，我们的拟合结果大体上吻合与原始数据。拟合得到的速率-能量参数空间直方图的值，也就是每个分区中的期望事例数，而原始直方图的值，就是每个分区的实测事例数。图6的结果也就显示了，当所试探的期望分布与实测分布最接近时，给出最优的试探结果。这一过程是似然函数估计方法的最大特点，正是基于此，它可以分辨这种在两个维度上的分布都非常接近，但是重叠区域较小的事件。

罗俊平的文章 [3] 中提到了使用主成分分析（PCA），也即对数据参数空间坐标矩阵进行奇异值分解，取最大的奇异值将数据降至一维，不过他没有给出具体的计算过程和结果，所以在图7中我们没有对比这个方法。但其实这个方法无非是对莫梁虹的方法的修正，依然是进行了降维操作，只是找到了一个降维之后数据点间隔最大的降维方向。而从图5(b) 中就可以看出，不论往哪个方向作投影，得到的数据都是具有相当大的重叠区域的，降维之后再使用一维的似然函数分析得到的结果与莫梁虹的方法大同小异。

5 多自由度对数似然函数拟合

我们前面的似然函数估计的过程中，都是考虑已知事件类型，分辨 A, B 两事件发生的次数，估计 N_A, N_B。而在实际的粒子物理实验中，还有另一种常见的工作是：从数据中寻找新的事件，确定新的事件的参数 μ, σ 等。这就需要解锁更多的自由度来做拟合。

我们对这种方法进行了尝试（见附录C.4），其物理过程依然与2一致。但是我们却没有得到比较理想的结果，即我们发现求解出来的似然函数极大值点非常依赖迭代初始值的选取。我们发现其原因是参数空间中存在太多的局部极小值，导致



(a) 原模拟赝事例　　(b) 拟合结果

图 6 速率-能量参数空间直方图的等高线

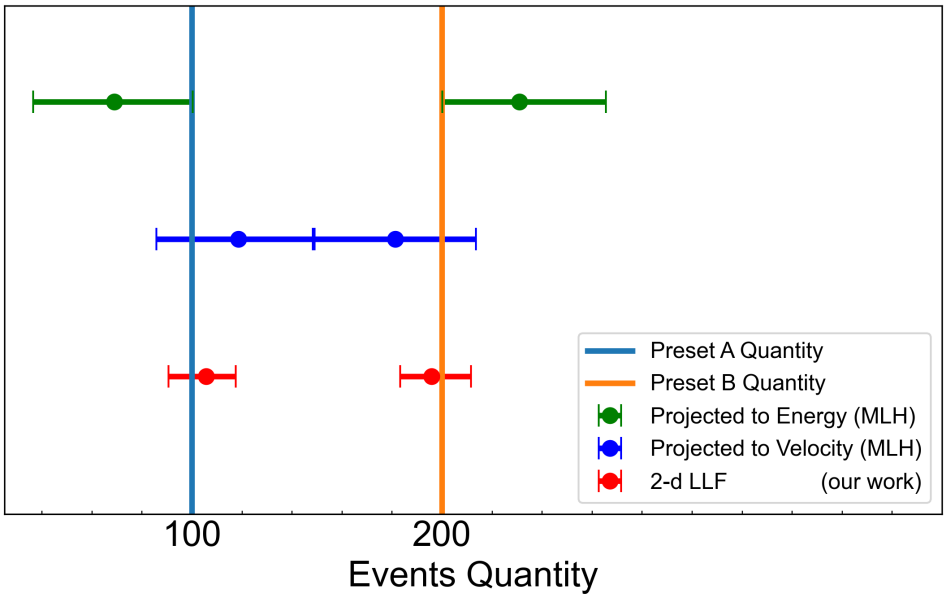


图 7 不同方法的分辨能力对比

结果无法收敛到真正的负对数似然函数的最小值。

这确实是极大似然估计作为一种成本函数很灵敏的优化方法存在的系统性问题，如果需要做更多自由度的拟合，需要对该方法进行进一步的“松弛”处理，以避免收敛到局部最优点的问题。

6 结论

我们采用极大似然估计的方法分辨两个随机生成的模拟赝事例的混合能量信号的事例数。我们基于中山大学基础物理实验教学团队的“对数似然函数分析”方法，做了进一步的优化和拓展，提出了“快速对数似然函数分析”和“多维对数似然函数分析”，使其计算效率、拟合的准确程度得到了很大的提高，拟合的不确定度得以降低。

声明与致谢

本工作在 Linux ubuntu_64 系统上完成。Linux 系统在数据处理方面具有模块化、内核通用性好、可扩展性强、开源等优点，使得我们的的工作很容易迁移到具体的实际粒子物理实验的数据处理过程中。

本工作的数值计算代码大部分都使用 Python 这一解释型、面向对象、动态的高级程序设计语言书写，并用交互式编程环境 Jupyter Notebook 进行调试。Python 中的 Numpy 和 SciPy 科学计算库对我们的工作提供了相当大的支持，Numpy 中函数的并行性使得我们的高精度大容量计算得以实现。

我们所提出的“快速对数似然函数分析”和“多维对数似然函数分析”都是基于中山大学基础物理实验教学团队所提供的实验讲义中原始的“对数似然函数分析”方法作出改进而完成的。在此对他们的工作致谢。

感谢王誉晨、莫梁虹、罗俊平对本工作的支持。

参考文献

[1] 黄臻成, 唐健, 沈韩. 中山大学物理学院基础物理实验讲义：实验 c10 粒子实验模拟数据分析. [EB/OL]. http://lovephysics.sysu.edu.cn/doku.php?id=courses:thirdlevel:exp10 (Accessed April 14, 2022).

[2] 莫梁虹. 基于 linux 和 python 的粒子物理实验数据分析. [EB/OL].

[3] 罗俊平. 基于 linux 的粒子实验模拟数据分析. [EB/OL].

[4] Samuel S Wilks. The large-sample distribution of the likelihood ratio for testing composite hypotheses. *The annals of mathematical statistics*, 9(1):60–62, 1938.

[5] J. A. Nelder and R. Mead. The downhill simplex method. *The Computer Journal*, 7:2–3, 1965.

[6] Gnu scientific library. [EB/OL]. https://www.gnu.org/software/gsl/ (Accessed April 10, 2022).

[7] Apple Inc. Apple open source. [EB/OL]. https://opensource.apple.com/source/Libm/Libm-2026/Source/Intel/ (Accessed April 10, 2022).

[8] Wikipedia of pi. [EB/OL]. https://en.wikipedia.org/wiki/Pi (Accessed April 10, 2022).

[9] LITTLEWOOD and E. J. Collected papers of srinivasa ramanujan. *Nature*, 123(3104):631–633, 1927.

[10] Intel Inc. Intel® 64 and ia-32 architectures software developer’s manual. [EB/OL]. https://www.felixcloutier.com/x86/ (Accessed April 10, 2022).

[11] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on ”Program Generation, Optimization, and Platform Adaptation”.

[12] Fftw. [EB/OL]. http://www.fftw.org/ (Accessed April 10, 2022).

[13] Shannon and E. C. Communication in the presence of noise. *Proceedings of the IEEE*, 1998.

Distinguishment and Fitting of Simulated Event Signals based on Maximum Likelihood Estimation

Yinkai Yu and Yuchen Wang
School of Physics, Sun Yat-sen University, Guangzhou 510275, China

Abstract: The data processing process of particle physics experiments often requires to distinguish the frequency and proportion of each event from the mixed signals of different known events. Logarithmic likelihood function analysis is an effective method to solve this problem. We propose a more efficient and fast logarithmic likelihood function analysis, and extend the logarithmic likelihood function analysis to a higher dimension, so that we can make use of the multi-dimensional data obtained from particle physics experiments. especially the multi-dimensional data with correlation to improve the accuracy of event resolution.

Key words: maximum likelihood estimation, particle physics experimental data processing, two-dimensional normal distribution, numerical stability

A 实验信息记录

A.1 基本信息

实验室房间号：202	实验人姓名（学号）：余荫铠（20343078）
实验桌桌号：206A01	合作者姓名（学号）：王誉晨（20343059）

A.2 实验环境

第一次实验（2022 年 3 月 23 日星期三下午）	室温：21	湿度：54
第一次实验（2022 年 3 月 30 日星期三下午）	室温：21	湿度：75
第一次实验（2022 年 4 月 6 日星期三下午）	室温：23	湿度：52

A.3 仪器设备

Linux 操作系统	ubuntu_64
编辑器	Jupyter Notebook、kate
编译器/解释器	python3.8.5、gcc、MinGW-w64
函数库	GSL、Numpy、Matplotlib、Scipy
服务器地址	ftp://lovephysics.sysu.edu.cn/Data /周三下午 B 组/C10 20343078 余荫铠 20340359 王誉晨/

B 思考题解答

题 1. 基于 *GSL* 提供的各类函数库，数值求解 π。

最简单的思路是利用

$$\pi = \arccos(-1) \tag{24}$$

可以使用 GSL[6] 的`gsl_complex gsl_complex_arccos_real(double x)`函数实现：

[in]:
```c
/* ------------------------------ */
/*             方法 1             */
/* ------------------------------ */
#include <iostream>
#include <gsl/gsl_complex.h>
#include <gsl/gsl_complex_math.h>

int main(){
    printf( "pi = %.6f", GSL_REAL( gsl_complex_arccos_real(-1) ) );
}
```

[out]:
```
pi = 3.141593
```

更进一步，我们查询 GSL`gsl_complex gsl_complex_arccos_real(double x)`函数的源代码，了解到它其实是基于 C 语言的 math 函数库的`acos()`函数，而 C 语言的 math 函数库在不同的系统中有不同的实现方式，我们在 Apple 的开源库 [7] 中了解到`acos()`函数是通过级数法计算的，并且 Apple 对近似区间

作了详尽的划分并预先算好了级数各项系数以优化计算速度。类似地我们也给出一个级数计算方法 [8-9]:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{(4k)!(1103+26390k)}{k!^4(396^{4k})} \tag{25}$$

[in]:
```
/* ------------------------------ */
/*             方法 2             */
/* ------------------------------ */
#include <iostream>
#include <gsl/gsl_sf.h>
#include <gsl/gsl_math.h>

int main(){
    double pi = 2*sqrt(2) * 1103 / 9801;
    for (int k = 1; k < 3; k++){            // 迭代两次
        pi += gsl_sf_fact(4 * k) * (1103 + 26390 * k) / \
            (gsl_pow_4(gsl_sf_fact(k)) * gsl_pow_int(396, 4 * k));
    }
    pi = 1 / pi;
    printf("pi = %.6f", pi);
    system("pause");
}
```

[out]:
```
pi = 3.141593
```

此外，在 Windows 的 Intel x86 或者 amd64 系统中，我们从《英特尔 ® 64 位和 IA-32 架构软件开发人员手册》中看到`acos()`则是利用数值积分实现的 [10]。在这里，我们可以更直接一点，跳过 $\pi=\arccos(-1)$ 而直接用数值积分来计算 π:

$$\pi=\int_{-1}^{1}2\sqrt{1-x^2}\mathrm{d}x \tag{26}$$

[in]:
```
/* ------------------------------ */
/*             方法 3             */
/* ------------------------------ */
#include <iostream>
#include <gsl/gsl_math.h>

double f(double x) {
    return 2 * sqrt(1 - x * x);
}

int main(){
    printf("pi = %.6f", f(1/sqrt(3)) + f(-1/sqrt(3)) );
    system("pause");
}
```

```
[out]: pi = 3.265986
```

当然，这里用最简单的两点 Gauss 公式作为示范，故对精度没有太高的要求。

题 2. *利用网络查找 fftw 库的函数使用方法。*

FFTW 是一个 C 语言子程序库 [11]，用于计算一维或多维、任意输入大小的离散傅立叶变换 (DFT)，支持实数和复数运算。我们在 FFTW 官网的手册 [12] 中了解了 FFTW 库的函数的基本使用方法。

FFTW 函数库的核心函数有四个：

```
/* 一维复数快速傅里叶变换： */
fftw_plan fftw_plan_dft_1d(int n, fftw_complex *in, fftw_complex *out, \
    int sign, unsigned flags);

/* 多维复数快速傅里叶变换： */
fftw_plan fftw_plan_dft(int rank, const int *n, fftw_complex *in,\
    fftw_complex *out, int sign, unsigned flags);

/* 一维实数快速傅里叶变换： */
fftw_plan fftw_plan_dft_r2c_1d(int n, double *in, fftw_complex *out, unsigned flags);

/* 多维实数快速傅里叶变换： */
fftw_plan fftw_plan_dft_r2c( int rank, const int *n, double *in, fftw_complex *out, \
    unsigned flags);
```

输入和输出的格式展示其中。它们的使用方法是类似的，因此下面我们就以其中的一维复数快速傅里叶变换`fftw_plan_dft_1d`函数为例说明它们的使用方法。

利用`fftw_plan_dft_1d`计算长度为 N 的一维 DFT 的代码通常的格式如下所示：

```
#include <fftw3.h>
...
{
    fftw_complex *in, *out;
    fftw_plan p;
    ...
    in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
    ...
    fftw_execute(p); /* repeat as needed */
    ...
    fftw_destroy_plan(p);
    fftw_free(in); fftw_free(out);
}
```

其中，`in`和`out`表示输入数据和输出数据的数组指针，N 为数据长度。在输入和输出的文件中，需要包含`fftw_complex`类型（定义为`double[2]`）数据的列表`in`和`out`，比如其中`in[i][0]`为第 i 个数据的实

部、in[i][1]为第 i 个数据的虚部。在进行变换之前，需要用fftw_alloc_complex(N)为数组分配内存，在计算完成后用fftw_free(in)释放内存。

函数 fftw_plan fftw_plan_dft(int rank, const int *n, fftw_complex*in, fftw_complex *out, int sign, unsigned flags)的第四个参数int sign用于控制傅里叶正变换或逆变换，FFTW_FORWARD(=-1) 表示正变换，FFTW_ACKWARD(=1) 表示逆变换。它们其实表示傅里叶变换因子的指数上的符号。第五个参数flags用于控制计算时间，通常用FFTW_MEASURE即可。需要注意的是，例程第 9 行的 plan 创建必须放在后续的数组赋值之前，否则FFTW_MEASURE会覆盖输入输出数组。

完成以上的初始化之后，使用void fftw_execute(const fftw_plan plan)函数执行傅里叶变换计划。该函数可以根据不同的输入输出数组重复使用。使用完之后用void fftw_destroy_plan(fftw_plan plan)释放内存。

题 3. *利用 Numpy 练习中初始的数组 arr, 产生一个 6×5×5 的三维数组, 在实验报告中编写一小段程序，实现将 arr 中尾数是 3 的数字替换为所在行的元素的总和。*

[in]:
```
import numpy as np

### 初始化
arr         = np.arange(1, 151)
arr.shape   = (2, 3, 5, 5)
musk  = arr%10==3
arr_1 = np.copy(arr)
arr_2 = np.copy(arr)

### 方法 1: 利用 where 函数
arr_1[musk] = np.sum(arr_1,axis=3)[np.where(musk)[0:3]]
# arr_1

### 方法 2: 利用 broadcast
arr_2[musk] = (np.sum(arr_2,axis=3)[:,:,:,None] * musk)[musk]
arr_2
```

[out]:
```
array([[[[  1,   2,  15,   4,   5],
         [  6,   7,   8,   9,  10],
         [ 11,  12,  65,  14,  15],
         [ 16,  17,  18,  19,  20],
         [ 21,  22, 115,  24,  25]],

        [[ 26,  27,  28,  29,  30],
         [ 31,  32, 165,  34,  35],
         [ 36,  37,  38,  39,  40],
         [ 41,  42, 215,  44,  45],
         [ 46,  47,  48,  49,  50]],

        [[ 51,  52, 265,  54,  55],
         [ 56,  57,  58,  59,  60],
         [ 61,  62, 315,  64,  65],
```

```
        [ 66,  67,  68,  69,  70],
        [ 71,  72, 365,  74,  75]]],


      [[[ 76,  77,  78,  79,  80],
        [ 81,  82, 415,  84,  85],
        [ 86,  87,  88,  89,  90],
        [ 91,  92, 465,  94,  95],
        [ 96,  97,  98,  99, 100]],

       [[101, 102, 515, 104, 105],
        [106, 107, 108, 109, 110],
        [111, 112, 565, 114, 115],
        [116, 117, 118, 119, 120],
        [121, 122, 615, 124, 125]],

       [[126, 127, 128, 129, 130],
        [131, 132, 665, 134, 135],
        [136, 137, 138, 139, 140],
        [141, 142, 715, 144, 145],
        [146, 147, 148, 149, 150]]]])
```

可见输出结果满足题目要求。

我这里的源代码写得比较精简，给出了两种方法，每个方法就一行，可读性可能不是很好，我这么写是因为题目要求“在规范的前提下，程序行越短越好”。

下面我对这两种方法做一下简单的解析。

初始化数组`musk`是一个形状一致于`arr`的布尔数组，在`arr`尾数是 3 的位置处，`musk`的数组元为`True`，否则为`False`。`arr_1`和`arr_2`通过`np.copy(arr)`来初始化其大小。

在方法一中，先看等号右边。`np.sum(arr_1,axis=3)`把原数组`arr_1`的第四个维度求和掉，即对行求和，返回的数组有三个维度。此外，`np.where(musk)`返回`musk`为`True`的坐标，每行对应原数组的一个维度，每列对应原数组的一个元素。那么，我们用`np.where(musk)`这个表示坐标的数组去索引`np.sum(arr_1,axis=3)`，注意到`np.sum(arr_1,axis=3)`相比于原数组`arr_1`，其第四个维度已经被求和掉了，因此我们只提取`np.where(musk)`的前三行切片`np.where(musk)[0:3]`（即`np.sum(arr_1,axis=3)`中满足条件的元素的坐标，该元素的值就是被求和的那一行的原数组元素的和），来索引`np.sum(arr_1,axis=3)`的值，返回一维数组储存`arr_1`中尾数是 3 的数字替换为所在行的元素的总和。等号左边是用`musk`索引`arr_1`，得到`arr_1`中尾数是 3 的元素组成的一维数组。这样等号左边是需要被替换的元素，等号右边是需要被替换为的新值，这样一个赋值语句就实现了题目要求。

再看方法二，它不需要使用`np.where`函数，只通过灵活运用 python 的数据结构就可以实现。等号右边的圆括号`()`里的数组的值为（为了方便读者理解，我先展示其结果，再解释其实现过程）：

[in]:
```
np.sum(arr_2,axis=3)[:,:,:,None] * musk
```

[out]:
```
array([[[[   0,    0,   27,    0,    0],
         [   0,    0,    0,    0,    0],
         [   0,    0,  117,    0,    0],
```

```
       [   0,    0,    0,    0,    0],
       [   0,    0,  207,    0,    0]],

      [[   0,    0,    0,    0,    0],
       [   0,    0,  297,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0,  387,    0,    0],
       [   0,    0,    0,    0,    0]],

      [[   0,    0,  477,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0,  567,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0,  657,    0,    0]]],


     [[[   0,    0,    0,    0,    0],
       [   0,    0,  747,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0,  837,    0,    0],
       [   0,    0,    0,    0,    0]],

      [[   0,    0,  927,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0, 1017,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0, 1107,    0,    0]],

      [[   0,    0,    0,    0,    0],
       [   0,    0, 1197,    0,    0],
       [   0,    0,    0,    0,    0],
       [   0,    0, 1287,    0,    0],
       [   0,    0,    0,    0,    0]]])
```

它的形状与原数组一致，`arr`中尾数是 3 的数字所在行的元素的总和被放在了需要被替换的位置。有了这个数组，其实接下来可行的操作就很多了，比如我们除了上面的方法二之外，还可以给出下面这种：其中`sums`就是我们上面展示的数组。这种思路是把原数组种`arr`中符合条件的元素置零，再把这个`sums`加在`arr`上。或者如方法二，用`musk`同时对`arr_2`和`sums`切片，把后者对应位置的置赋到前者的对应位置。

最后我们来看看`sums`这个数组怎么通过广播（broadcast）来实现。我们知道`np.sum(arr_2,axis=3)`相比于原数组`arr_2`，其第四个维度已经被求和掉了，所以我们用`[:,:,:,None]`来切片，以给其扩充第四个维度但赋空值，这样`np.sum(arr_2,axis=3)[:,:,:,None]`的维数就和原数组`arr_2`一致了。接下来我们将它与`musk`，这一步就利用了广播操作，填充了其第四个维度的值，并且利用`musk`本身的布尔值来使其只有满足条件的行的和得到保留，这就生成了我们需要的数组`sums`。

题 4. *用 pandas 中的数据，利用统计学相关性函数，分析不同学科成绩之间的是否存在 Pearson 相关性？*

这个题不需要像讲义上那样，用 numpy 中的统计学相关性函数，比如计算皮尔逊相关系数的函数`np.corrcoef`。如果要用这个函数，还需要进行一定的数据清洗，调整 data 的数据结构。我们有更简洁的实现途径，即利用强大的`pandas`库中`pandas.core.frame.DataFrame`数据类型的`corr()`方法，可以一步到位，避免重复造轮子：

[in]:
```
import pandas as pd

# 读取数据
path = r"C10.2data.CSV"
data = pd.read_csv(path,sep=",", header=0, index_col=0, encoding='GBK')

# 计算相关系数
data.corr()
```

[out]:

	Math	English	Physics
Math	1.000000	0.011579	0.870921
English	0.011579	1.000000	0.088289
Physics	0.870921	0.088289	1.000000

由此可见，数学成绩和物理成绩之间具有较强的线性相关性，其皮尔逊相关系数为 0.870921，为正相关。这意味着数学成绩好的同学很大概率物理成绩也好，物理成绩好的同学很大概率数学成绩也好。此外，其他学科之间没有明显的线性相关性，不过至少也没有出现负相关，这意味着不会有哪些学科之间的成绩会出现相互抑制的作用。

题 5. 请解释利用快速傅里叶变换 *(fft)* 实现分析的程序思路。

[in]:
```
from matplotlib import pyplot as plt
import matplotlib
import numpy as np
from scipy.fftpack import fft
```

在 python 中实现 fft 需要引入`scipy`库中的`scipy.fftpack.ff`函数。

我们举一个具体的例子来展现思路。下面分析一个随时间衰减的振荡信号

$$\text{signals} = \mathrm{e}^{-\frac{t}{2}} \sin(2\pi t) \tag{27}$$

显然其频率为 1 Hz，衰减时间为 2 s。根据香农采样定理 [13]，采样频率需要大于 2 Hz。下面我们对这个信号进行采样，时域长度为 10 s，采样点数为 200，则采样频率为 20，符合香农采样定理的条件。

[in]:
```
# 随时间衰减的振荡时域信号采样
T       = 10   # 时域长度
N       = 200  # 采样点数
t       = np.linspace(0, T, N, endpoint=False)
signals = np.exp(-0.5*t) * np.sin(2*np.pi*t)
```

`fft(signals)`即返回对`signals`数组进行快速傅里叶变换后的结果。我们需要对`signals_fft`进行下面的处理。首先，我们只关系振幅谱，因此我们对`signals_fft`取模。此外，`signals_fft`本身是采样的求和，为了得到频谱分布，我们要对其进行归一化处理，除以采样点数 N。由于正频率和负频率在物理意

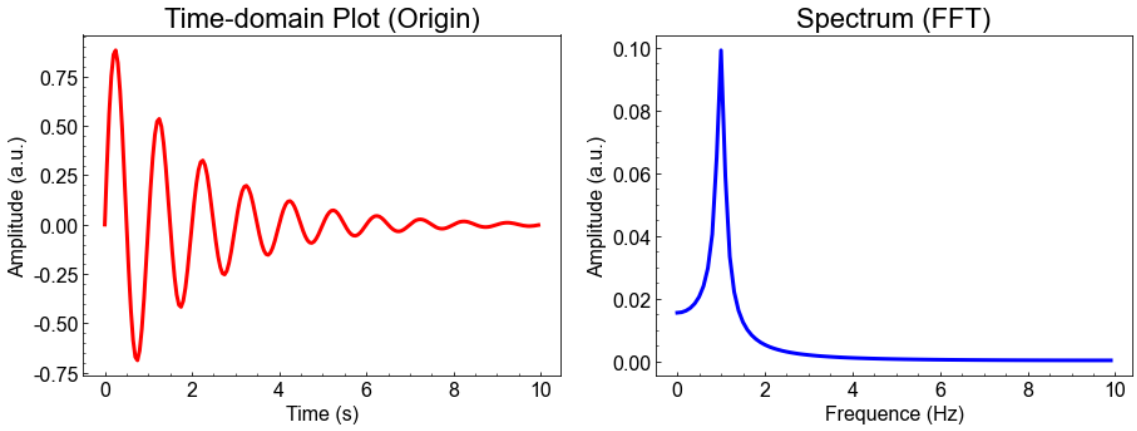
义上是对称的，我们只关心正的一半就好，因此我们对其进行取半，得到频谱signals_f。

[in]:
```
# fft
signals_fft     = fft(signals)
signals_fft_abs = np.abs(signals_fft) # 振幅谱
# signals_fft_agl = np.angle(fft_y)   # 相位谱

# 归一化和半处理
f          = np.fft.fftfreq(N, 10/N)[:int(N/2)]
signals_f = (signals_fft_abs / N)[:int(N/2)]
```

将 fft 变换前后的信号可视化：

[in]:
```
## 可视化
matplotlib.rcParams['figure.figsize']      = [15, 5]

# 时域图
plt.subplot(121)
plt.plot(t, signals,   linewidth=3, c='r')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (a.u.)')
plt.title('Time-domain Plot (Origin)',fontsize=22)

# 频谱图
plt.subplot(122)
plt.plot(f, signals_f, linewidth=3, c='b')
plt.xlabel('Time (s)')
plt.ylabel('Frequence (a.u.)')
plt.title('Spectrum (FFT)',fontsize=22)

plt.show()
```

[out]:



可见 fft 结果的频谱峰确实如我们所设的，为 1 Hz。

题 6. 什么是对数似然函数法？分别利用极大似然函数法和对数似然函数法分析数据，比较两种方法所得结果的异同。

对数似然函数法是指对似然函数值取负对数，将似然函数求极大值点的问题转化为求其负对数的极小值点的问题。

使用对数似然函数对本实验的数据进行分析，仍然可以拟合出极大似然点，一致于 (11) 所给的拟合中心值，而其数值的病态则体现在置信区间的求解中。事实上，采用对数似然函数分析，得到的 $l(N_A, N_B)$ 极小值为

$$\min l = l(N_A^*, N_B^*) = -469.38 \tag{28}$$

而采用直接似然函数分析得到的 $L(N_A, N_B)$ 极大值为

$$\max L = L(N_A^*, N_B^*) = 8.3728 \times 10^{-32} \tag{29}$$

后者处于相当小的数量级——而在求解置信区间时，我们要求

$$\chi^2 = -2(l'(\theta) - l_{\min}) = -2 \ln \frac{L'(\theta)}{L_{\max}} = 1 \tag{30}$$

此时对于直接似然函数分析的结果，需要计算 $L'(\theta)$ 和 $L_{\max}$ 两个很小的数的比值，这是一个典型的数值病态问题。即使这时再将 $\ln \frac{L'(\theta)}{L_{\max}}$ 化为两个对数值的差来计算，仍不能避免其病态性，因为对数函数对很小的自变量的误差是相当敏感的。

对数似然函数分析和似然函数分析的具体过程如下（这里对两个方法进行了高度的封装，以便后续思考题的使用）：

[2]:
```
# package
import numpy as np
from scipy.stats import poisson
import scipy.optimize as opt
import matplotlib.pyplot as plt
import matplotlib
```

[3]:
```
# preset of plot
# plt.rcParams['savefig.dpi']          = 600
# plt.rcParams['figure.dpi']           = 600       # 输出
plt.rcParams['figure.dpi']           = 100       #  草稿
plt.rcParams['figure.figsize']       = [5, 5]
# plt.rcParams['xtick.top']            = True
plt.rcParams['xtick.direction']      = 'in'
plt.rcParams['xtick.minor.visible'] = True
# plt.rcParams['ytick.right']          = True
plt.rcParams['ytick.direction']      = 'in'
plt.rcParams['ytick.minor.visible'] = True
plt.rcParams['font.size']            = 19
# plt.rcParams['font.family']          = ['DejaVu Serif']
# plt.rcParams['font.family']          = ['sans-serif']
# plt.rcParams['font.sans-serif']      = ['SimHei']
# plt.rcParams['font.family']          = ['DengXian']
plt.rcParams['font.family']          = ['Arial']
plt.rcParams['mathtext.default']     = 'regular'
```

```
plt.rcParams['errorbar.capsize']    = 3
plt.rcParams['figure.facecolor']    = (1,1,1)
plt.rcParams['lines.linewidth']     = 3
```

预封装

[4]:
```
def class_a(nev):
    return np.random.normal(10, 2, size=nev)
def class_b(nev):
    return np.random.normal(15, 3, size=nev)

mc_class_a = class_a(1000000)
mc_class_b = class_b(1000000)

def set_data(a_num, b_num):
    return np.concatenate([class_a(a_num), class_b(b_num)])

def set_bins(data, bin_num):
    data_min = min(data) * 0.999
    data_max = max(data) * 1.001
    binning = np.linspace(data_min, data_max, bin_num)
    return binning
```

[5]:
```
class LikelihoodFunction:

    def __init__(self, data, event_classes, binning):
        self.data_counts = np.histogram(data, bins=binning)[0]
        self.class_pdfs  = []
        for event_class in event_classes:
            pdf_counts = np.histogram(event_class, bins=binning)[0]
            pdf_norm   = pdf_counts / np.sum(pdf_counts)
            self.class_pdfs.append(pdf_norm)

    def __call__(self, *params):
        observed  = self.data_counts
        expecteds = [ scale * pdf for scale, pdf in zip(params, self.class_pdfs) ]
        expected  = np.sum(expecteds,axis=0)
        bin_probabilities = poisson.pmf(observed, expected)
        return np.prod(bin_probabilities)

class NegativeLogLikelihoodFunction:

    def __init__(self, data, event_classes, binning):
        # initialize likelihood function
        self.data_counts = np.histogram(data, bins=binning)[0]
```

```
        self.class_pdfs  = []
        for event_class in event_classes:
            pdf_counts = np.histogram(event_class, bins=binning)[0]
            pdf_norm   = pdf_counts / np.sum(pdf_counts)
            self.class_pdfs.append(pdf_norm)

    def __call__(self, *params):
        observed  = self.data_counts
        expecteds = [ scale*pdf for scale,pdf in zip(params,self.class_pdfs) ]
        expected  = np.sum(expecteds,axis=0)
        mask      = expected > 0
        bin_nlls  = expected[mask] - observed[mask] * np.log(expected[mask])
        return np.sum(bin_nlls)
```

[6]:
```
def nlog_opt(data, nllfn, num_0):
    nll_result = opt.minimize(
        lambda x: nllfn(*x),
        x0     = num_0,
        method = 'Nelder-Mead',
    )
    return nll_result

def nake_opt(data, lfn, num_0):
    result = opt.minimize(
        lambda x: -lfn(*x),
        x0     = (150, 150),
        method = 'Nelder-Mead',
    )
    return result
```

[37]:
```
class profile_nlog:
    def __init__(self, clas, func, mini, n_0=50):
        self.c = clas
        self.f = func
        self.m = mini
        self.n = n_0
    def __call__(self, *params):
        nev    = params
        if self.c == 'a':
            return opt.minimize(lambda x: self.f(nev,x[0]), x0=(self.n,),␣
 ↪method='Nelder-Mead').fun - self.m
        if self.c == 'b':
            return opt.minimize(lambda x: self.f(x[0],nev), x0=(self.n,),␣
 ↪method='Nelder-Mead').fun - self.m
```

```
class profile_nake:
    def __init__(self, clas, func, mini, n_0=50):
        self.c = clas
        self.f = func
        self.m = mini
    def __call__(self, *params):
        nev    = params
        if self.c == 'a':
            L  = self.m
            L_ = opt.minimize(lambda x: self.f(nev,x[0]), x0=(n_0,),␣
 ↪method='Nelder-Mead').fun
            return -2 * ( np.log(L_) - np.log(L) )
        if self.c == 'b':
            L  = self.m
            L_ = opt.minimize(lambda x: self.f(x[0],nev), x0=(n_0,),␣
 ↪method='Nelder-Mead').fun
            return -2 * ( np.log(L_) - np.log(L) )

def confidence_interval(delta_nll_fn, central, step):
    lo = opt.brentq(lambda x: delta_nll_fn(x)-0.5, central-step, central)
    hi = opt.brentq(lambda x: delta_nll_fn(x)-0.5, central, central+step)
    return lo, hi

def err_range(func, result, ftype):
    if ftype == 'nlog':
        profile_class_a = profile_nlog('a', func, result.fun)
        profile_class_b = profile_nlog('b', func, result.fun)
    else:
        profile_class_a = profile_nake('a', func, result.fun)
        profile_class_b = profile_nake('b', func, result.fun)
    central_a       = result.x[0]
    lo_a, hi_a      = confidence_interval(profile_class_a, central_a, 0.5*sum(result.
 ↪x))
    central_b       = result.x[1]
    lo_b, hi_b      = confidence_interval(profile_class_b, central_b, 0.5*sum(result.
 ↪x))
    return lo_a, hi_a, lo_b, hi_b
```

[8]:
```
def Likeli_Solve(
    data=set_data(100, 200),
    bin_num=40,
    likelihood='nlog',
    num_0=(150, 150),
):
```

```
    # 最终的封装函数
    binning = set_bins(data, bin_num)
    if likelihood == 'nlog':
        func   = NegativeLogLikelihoodFunction(data,[mc_class_a,mc_class_b],binning)
        result = nlog_opt(data, func, num_0)
        lo_a, hi_a, lo_b, hi_b = err_range(func, result, 'nlog')
        num_a = result.x[0]
        num_b = result.x[1]
        return num_a, num_b, lo_a, hi_a, lo_b, hi_b
    else:
        func   = LikelihoodFunction(data, [mc_class_a, mc_class_b], binning)
        result = nake_opt(data, func, num_0)
        # lo_a, hi_a, lo_b, hi_b = err_range(func, result, 'nake')
        num_a = result.x[0]
        num_b = result.x[1]
        return num_a, num_b
```

极大似然函数法和对数似然函数法

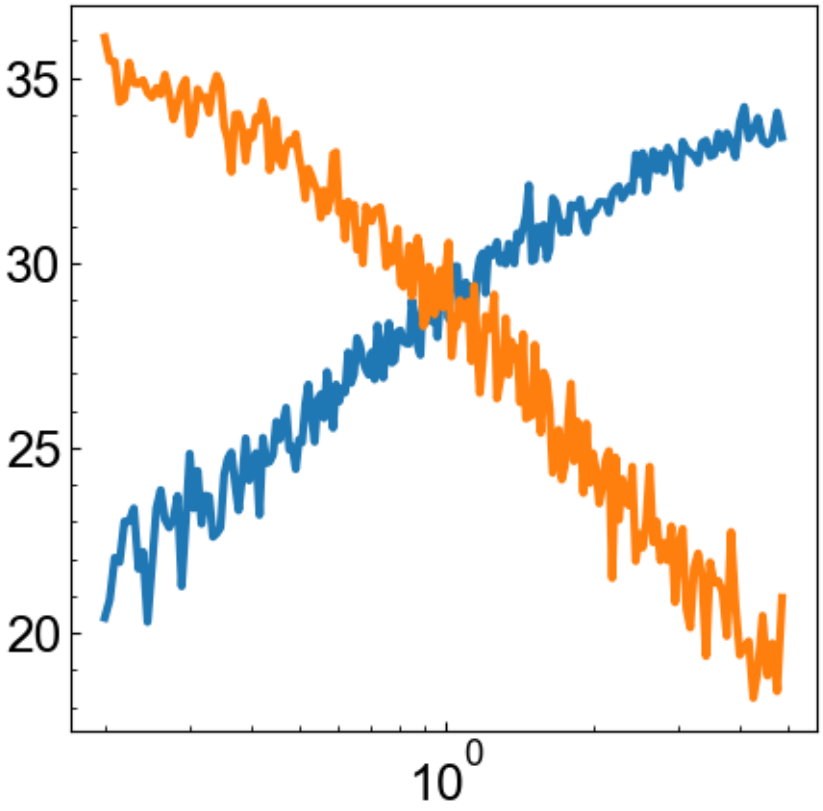
[34]:
```
# 生成模拟信号
data = set_data(100, 200)

# 极大似然函数分析
print(Likeli_Solve(data=data, likelihood='nake', num_0=(150, 150)))

# 对数似然函数分析
print(Likeli_Solve(data=data, likelihood='nlog', num_0=(150, 150)))
```

```
(102.5237788591699, 197.4761829508451)
(102.5237788591699, 197.4761829508451, 90.00175791427556, 115.77439364802686,
181.714967789164, 214.0268507620153)
```

题 7. 调整 A、B 随机赝事件的比例，但仍采用对数似然函数法进行分析，探讨保持相同置信水平的情况下，置信度区间如何变化。

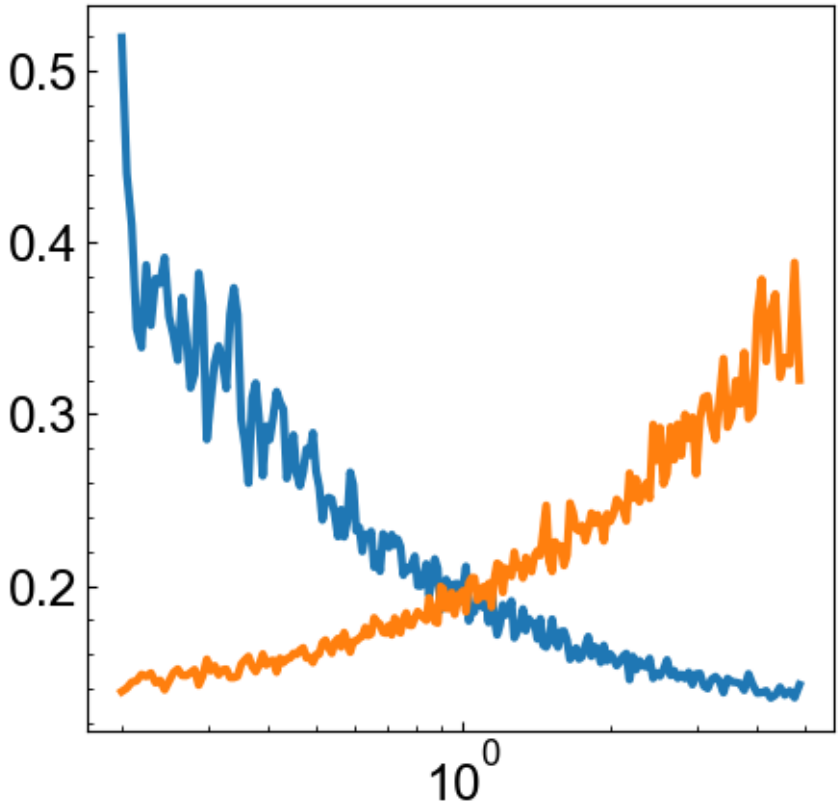
[22]:
```
# 置信区间
ratio = np.arange(50, 250) / np.arange(250, 50, -1)
plt.axes(xscale='log')
plt.plot(ratio, ran_a)
plt.plot(ratio, ran_b)
```

[22]:
```
[<matplotlib.lines.Line2D at 0x303dbd60>]
```



[30]:
```
# 相对误差
plt.axes(xscale='log')
plt.plot(ratio, rerr_a)
plt.plot(ratio, rerr_b)
```

[30]: [<matplotlib.lines.Line2D at 0x30855478>]



N_A^* 的置信区间随着 N_A/N_B 的增大而增大，N_B^* 的置信区间变化趋势相反。实际上有物理意义的是 N_A^*, N_B^* 的相对估计误差，即相对置信区间。N_A^* 的相对置信区间随着 N_A/N_B 的增大而减小，N_B^* 的相对估计误差变化趋势相反。

题 8. 调整 A、B 随机事件的总量，但保持原来的比例不变，探讨置信度区间的变化情况。

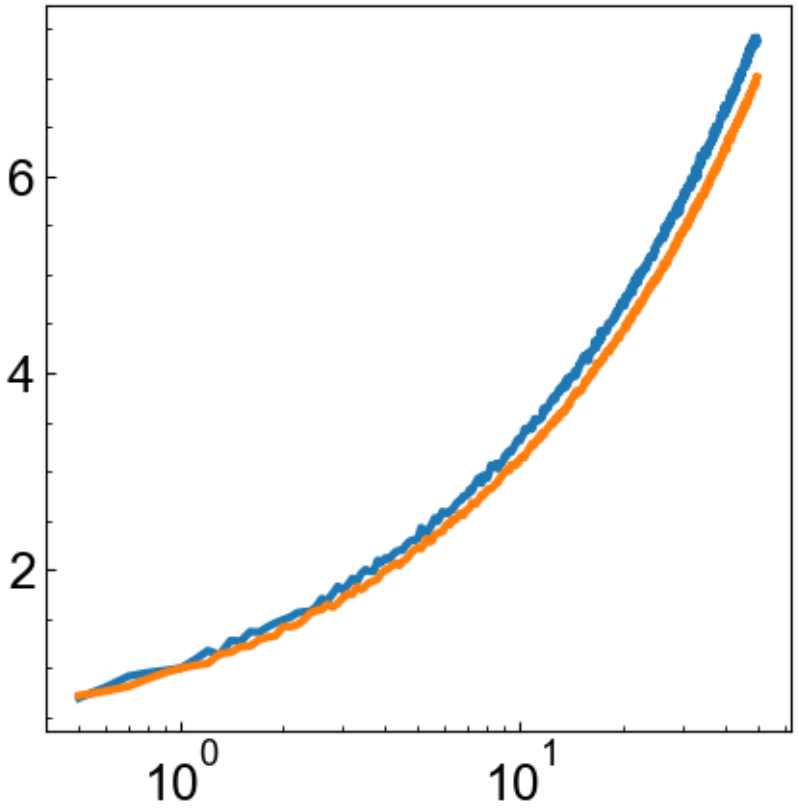
改变事件总量

[38]:
```
ran_a  = []
ran_b  = []
rerr_a = []
rerr_b = []

for scr in np.arange(0.5, 50, 0.1):

    a    = int(100*scr)
    b    = int(200*scr)
    data = set_data(a, b)
    num_a, num_b, lo_a, hi_a, lo_b, hi_b = Likeli_Solve(data=data,␣
↪likelihood='nlog', num_0=(150*scr, 150*scr))

    lenc_a = hi_a - lo_a
    lenc_b = hi_b - lo_b
```

```
    ran_a.append(lenc_a)
    ran_b.append(lenc_b)

    r_a    = lenc_a / num_a
    r_b    = lenc_b / num_b
    rerr_a.append(r_a)
    rerr_b.append(r_b)

    print(r_a)
```

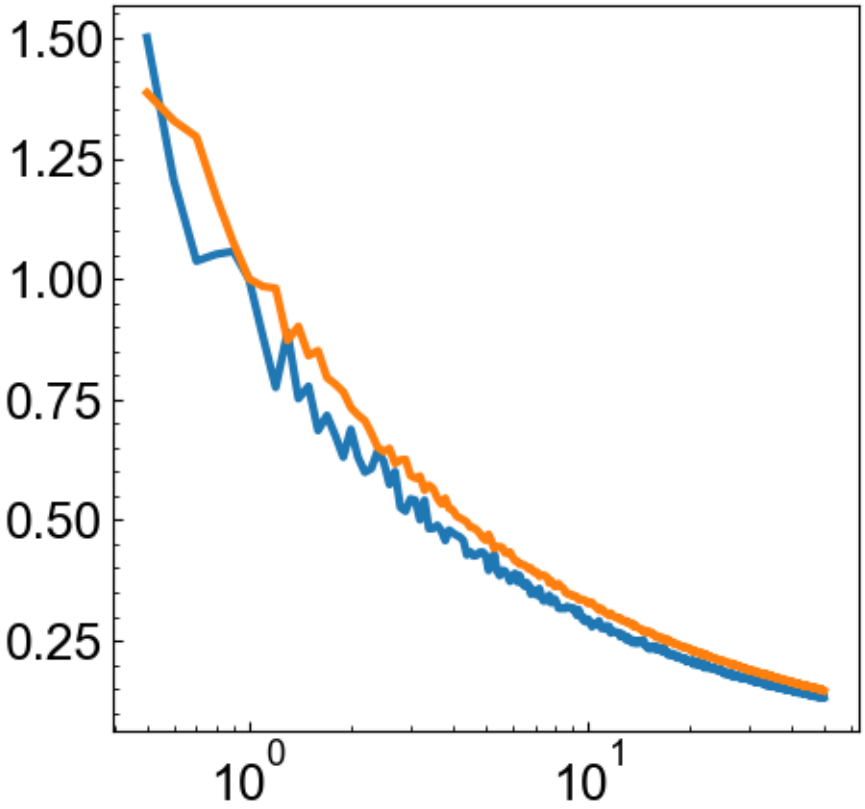
[48]:
```
# 置信区间

scale   = np.arange(0.5, 50, 0.1)
ran_a_0 = ran_a[5]
ran_b_0 = ran_b[5]
rer_a_0 = rerr_a[5]
rer_b_0 = rerr_b[5]
ran_a   = np.array(ran_a)   / ran_a_0
ran_b   = np.array(ran_b)   / ran_b_0
rerr_a  = np.array(rerr_a)  / rer_a_0
rerr_b  = np.array(rerr_b)  / rer_b_0

plt.axes(xscale='log')
plt.plot(scale, ran_a)
plt.plot(scale, ran_b)
```

[48]:
```
[<matplotlib.lines.Line2D at 0x30397418>]
```



[49]:
```
# 相对误差
plt.axes(xscale='log')
plt.plot(scale, rerr_a)
plt.plot(scale, rerr_b)
```

[49]: [<matplotlib.lines.Line2D at 0x3282da00>]



总量越大，置信区间越大，当然这是一个平凡的结论。有物理意义的是其相对置信区间下降。

题 9. *实际的粒子实验原始数据中，除了能量外，还有许多其他的信息，如速率等，如何利用两个维度的相关关系，提高区分 A、B 两个事件的分辨能力？*

在正文4多维对数似然函数分析部分有非常详细的分析了。

C 代码

C.1 基于极大似然估计的事件分辨

见B。

C.2 快速对数似然分析

快速对数似然函数法

[143]:
```
data=set_data(100, 200)
bin_num=40
likelihood='nlog'
num_0=(150, 150)

nllfn = NegativeLogLikelihoodFunction(data,[mc_class_a,mc_class_b],binning)
def nllfn_1_free(a_num):
    b_num = len(data) - a_num
    return nllfn(a_num, b_num)
```

```
result1 = opt.minimize(
    lambda x: nllfn_1_free(x),
    x0     = num_0[0],
    method = 'Nelder-Mead',
)
print(result1)

result2 = opt.minimize(
    lambda x: nllfn(*x),
    x0     = num_0,
    method = 'Nelder-Mead',
)
print(result2)

result3 = opt.minimize(
    lambda x: nllfn(*x),
    x0     = num_0,
    method = 'SLSQP',
    constraints = ({'type': 'eq', 'fun': lambda x: sum(x)- len(data)})
)
print(result3)

result4 = opt.minimize(
    lambda x: nllfn(*x),
    x0     = num_0,
    method = 'SLSQP',
)
print(result4)
```

```
 final_simplex: (array([[93.38018417],
       [93.38012695]]), array([-371.85912395, -371.85912395]))
           fun: -371.8591239509447
       message: 'Optimization terminated successfully.'
          nfev: 44
           nit: 22
        status: 0
       success: True
             x: array([93.38018417])
 final_simplex: (array([[ 93.06891834, 205.93105442],
       [ 93.06891339, 205.93114769],
       [ 93.06895271, 205.93106822]]), array([-371.86079247, -371.86079247,
-371.86079247]))
           fun: -371.86079247255327
       message: 'Optimization terminated successfully.'
```

```
         nfev: 93
          nit: 47
       status: 0
      success: True
            x: array([ 93.06891834, 205.93105442])
    fun: -371.8591239442911
    jac: array([0.00327301, 0.00336838])
message: 'Optimization terminated successfully'
   nfev: 19
    nit: 6
   njev: 6
 status: 0
success: True
      x: array([ 93.38149052, 206.61850948])
    fun: -371.86079246684886
    jac: array([1.14440918e-05, 7.62939453e-06])
message: 'Optimization terminated successfully'
   nfev: 42
    nit: 14
   njev: 14
 status: 0
success: True
      x: array([ 93.06931361, 205.93251728])
```

可以看出迭代次数大大减小，效率提高了不少。对于有约束的算法和无约束的算法，细节上稍有不同。下面我们对比一下快速对数似然函数法的效率增益比

[181]:
```
mthds_nocons = {
    'Powell',
    'BFGS',
    'CG',
    'TNC',
    'COBYLA',
    'Nelder-Mead',
    'L-BFGS-B'
}

mthds_cons = {
    'trust-constr',
    'SLSQP'
}

mthds_dl   = {
    'Newton-CG',
    'trust-ncg',
```

```
    'trust-krylov',
    'trust-exact',
    'dogleg'
}

print('算法\t优化前\t优化后\t增益比')

for mthd in mthds_nocons:

    result_f   = opt.minimize(
        lambda x: nllfn(*x),
        x0     = num_0,
        method = mthd,
    )

    result_l   = opt.minimize(
        lambda x: nllfn_1_free(x),
        x0     = num_0[0],
        method = mthd,
    )

    if result_f.success * result_l.success == True :
        print("{}\t{}\t{}\t{:.2f}%".format(mthd, result_f.nfev, result_l.nfev,␣
 ↪result_f.nfev/result_l.nfev*100))

for mthd in mthds_cons:

    result_f   = opt.minimize(
        lambda x: nllfn(*x),
        x0     = num_0,
        method = mthd,
        constraints = ({'type': 'eq', 'fun': lambda x: sum(x)- len(data)}),
    )

    result_l   = opt.minimize(
        lambda x: nllfn_1_free(x),
        x0     = num_0[0],
        method = mthd,
        constraints = ({'type': 'eq', 'fun': lambda x: sum(x)- len(data)}),
    )

    if result_f.success * result_l.success == True :
        print("{}\t{}\t{}\t{:.2f}%".format(mthd, result_f.nfev, result_l.nfev,␣
 ↪result_f.nfev/result_l.nfev*100))
```

算法	优化前	优化后	增益比
BFGS	54	16	337.50%
Powell	75	22	340.91%
CG	120	84	142.86%
Nelder-Mead	93	44	211.36%
COBYLA	113	80	141.25%
L-BFGS-B	30	14	214.29%
TNC	207	88	235.23%
SLSQP	19	4	475.00%
trust-constr	27	10	270.00%

C.3 多维对数似然分析

[1]:
```python
# package
import numpy as np
from scipy.stats import poisson
import scipy.optimize as opt
import matplotlib.pyplot as plt
import matplotlib
```

[2]:
```python
# preset of plot
# plt.rcParams['savefig.dpi']          = 600
# plt.rcParams['figure.dpi']           = 600      # 输出
plt.rcParams['figure.dpi']           = 100      #  草稿
plt.rcParams['figure.figsize']       = [5, 5]
# plt.rcParams['xtick.top']            = True
plt.rcParams['xtick.direction']      = 'in'
plt.rcParams['xtick.minor.visible'] = True
# plt.rcParams['ytick.right']          = True
plt.rcParams['ytick.direction']      = 'in'
plt.rcParams['ytick.minor.visible'] = True
plt.rcParams['font.size']            = 19
# plt.rcParams['font.family']          = ['DejaVu Serif']
# plt.rcParams['font.family']          = ['sans-serif']
# plt.rcParams['font.sans-serif']      = ['SimHei']
# plt.rcParams['font.family']          = ['DengXian']
plt.rcParams['font.family']          = ['Arial']
plt.rcParams['mathtext.default']     = 'regular'
plt.rcParams['errorbar.capsize']     = 3
plt.rcParams['figure.facecolor']     = (1,1,1)
plt.rcParams['lines.linewidth']      = 3
```

预封装

[3]:
```python
class EventClass_2d:
```

```
    def __init__(self, E_mu, E_sigma, v_mu, v_sigma, rho_2d):
        m1  = E_mu
        m2  = v_mu
        s1  = E_sigma
        s2  = v_sigma
        r12 = rho_2d
        self.mean = [m1, m2]
        self.cov  = [ [s1**2, r12*s1*s2], [r12*s1*s2, s2**2] ]

    def __call__(self, *params):
        return np.random.multivariate_normal(self.mean, self.cov, size=params)

def set_class_2d(E_mu, E_sigma, v_mu, v_sigma, rho_2d):
    class_a = EventClass_2d(E_mu[0], E_sigma[0], v_mu[0], v_sigma[0], rho_2d[0])
    class_b = EventClass_2d(E_mu[1], E_sigma[1], v_mu[1], v_sigma[1], rho_2d[1])
    return class_a, class_b

def set_data(a_num, b_num):
    return np.concatenate([class_a(a_num), class_b(b_num)])

def set_bins(data, bin_num):
    E      = data[:,0]
    v      = data[:,1]
    E_bin  = bin_num[0]
    v_bin  = bin_num[1]
    E_min  = min(E)
    E_max  = max(E)
    v_min  = min(v)
    v_max  = max(v)
    E_step = ( E_max - E_min ) / E_bin
    v_step = ( v_max - v_min ) / v_bin
    E_binning  = np.linspace(E_min-E_step, E_max+E_step, E_bin)
    v_binning  = np.linspace(v_min-v_step, v_max+v_step, v_bin)
    return E_binning, v_binning

def bin_data(data, E_binning, v_binning):
    E      = data[:,0]
    v      = data[:,1]
    data_counts = []
    for i in range(len(E_binning)-1):
        for j in range(len(v_binning)-1):
            # 这里的布尔数组相乘效率太低了！
            # data_count = np.sum( (E>E_binning[i]) * (E<E_binning[i+1]) *␣
↪(v>v_binning[j]) * (v<v_binning[j+1]) )
```

```
            data_count = np.sum( np.prod([(E>E_binning[i]), (E<E_binning[i+1]),␣
 ↪(v>v_binning[j]), (v<v_binning[j+1])], axis=0) )
            data_counts.append( data_count )
        if i%5==0:
            print('{}%'.format((i+1)*100/38))
    print('100%')
    return data_counts


def bin_pdf(class_e, E_binning, v_binning):
    num         = 1000000
    data        = class_e(num)
    data_counts = bin_data(data, E_binning, v_binning)
    pdf_counts  = np.array(data_counts) / num
    return pdf_counts
```

[4]:
```
class NegativeLogLikelihoodFunction:
    def __init__(self, data_counts, pdf_counts_a, pdf_counts_b):
        self.data   = data_counts
        self.pdf_a  = pdf_counts_a
        self.pdf_b  = pdf_counts_b
    def __call__(self, a_num, b_num):
        observed  = np.array(self.data)
        expected  = np.array( a_num * self.pdf_a + b_num * self.pdf_b )
        mask      = expected > 0
        bin_nlls  = expected[mask] - observed[mask] * np.log(expected[mask])
        return np.sum(bin_nlls)
```

[5]:
```
def nlog_opt(nllfn, num_0):
    nll_result = opt.minimize(
        lambda x: nllfn(*x),
        x0     = num_0,
        method = 'Nelder-Mead',
    )
    return nll_result

class profile_nlog:
    def __init__(self, clas, func, mini):
        self.c = clas
        self.f = func
        self.m = mini
    def __call__(self, *params):
        nev    = params
        if self.c == 'a':
```

```
            return opt.minimize(lambda x: self.f(nev,x[0]), x0=(50,),␣
 ↪method='Nelder-Mead').fun - self.m
        if self.c == 'b':
            return opt.minimize(lambda x: self.f(x[0],nev), x0=(50,),␣
 ↪method='Nelder-Mead').fun - self.m

def confidence_interval(delta_nll_fn, central, step):
    lo = opt.brentq(lambda x: delta_nll_fn(x)-0.5, central-step, central)
    hi = opt.brentq(lambda x: delta_nll_fn(x)-0.5, central, central+step)
    return lo, hi

def err_range(func, result):
    profile_class_a = profile_nlog('a', func, result.fun)
    profile_class_b = profile_nlog('b', func, result.fun)
    central_a       = result.x[0]
    lo_a, hi_a      = confidence_interval(profile_class_a, central_a, 50)
    central_b       = result.x[1]
    lo_b, hi_b      = confidence_interval(profile_class_b, central_b, 50)
    return lo_a, hi_a, lo_b, hi_b
```

[6]:
```
def Likeli_Solve(data_counts, pdf_counts_a, pdf_counts_b, num_0):
    func   = NegativeLogLikelihoodFunction(data_counts, pdf_counts_a, pdf_counts_b)
    result = nlog_opt(func, num_0)
    lo_a, hi_a, lo_b, hi_b = err_range(func, result)
    num_a = result.x[0]
    num_b = result.x[1]
    return num_a, num_b, lo_a, hi_a, lo_b, hi_b
```

高维对数似然函数法

[138]:
```
# 参数

E_mu    = [14, 15]
E_sigma = [ 2,  3]
# v_mu    = [0.8/np.sqrt(3), 2/np.sqrt(21)]
v_mu    = [0.45, 0.48]
v_sigma = [0.08, 0.05]
rho_2d  = [0.99, 0.99]

class_a, class_b = set_class_2d(E_mu, E_sigma, v_mu, v_sigma, rho_2d)

bin_num = [20, 20]
num_0   = (150, 150)
```

[115]:
```
# # 观察分布
# rho_2d  = [0, 0]
# class_a, class_b = set_class_2d(E_mu, E_sigma, v_mu, v_sigma, rho_2d)
# plt.rcParams['savefig.dpi']         = 600
# plt.rcParams['figure.dpi']          = 600       # 输出
# plt.rcParams['figure.figsize']      = [7, 7]
# test = class_a(100)
# plt.scatter(test[:,0], test[:,1])
# test = class_b(200)
# plt.scatter(test[:,0], test[:,1])
# plt.xlabel('Energy (MeV)')
# plt.ylabel('Velocity (c)')
# plt.legend(['Event A', 'Event B'])
# plt.savefig('img/二维正态 r=0.png')
```

[116]:
```
# 观察分布
# rho_2d  = [0.99, 0.99]
# class_a, class_b = set_class_2d(E_mu, E_sigma, v_mu, v_sigma, rho_2d)
# plt.rcParams['savefig.dpi']         = 600
# plt.rcParams['figure.dpi']          = 600       # 输出
# plt.rcParams['figure.figsize']      = [7, 7]
# test = class_a(100)
# plt.scatter(test[:,0], test[:,1])
# test = class_b(200)
# plt.scatter(test[:,0], test[:,1])
# plt.xlabel('Energy (MeV)')
# plt.ylabel('Velocity (c)')
# plt.legend(['Event A', 'Event B'])
# plt.savefig('img/二维正态 r=0.99.png')
```

[139]:
```
# 生成待分析数据

data = set_data(100, 200)
```

[140]:
```
# 分区

E_binning, v_binning = set_bins(data, bin_num)
data_counts = bin_data(data, E_binning, v_binning)
```

```
2.6315789473684212%
15.789473684210526%
28.94736842105263%
42.10526315789474%
100%
```

[54]:
```
# data_counts
```

[141]:
```
# 计算 pdf
pdf_counts_a = bin_pdf(class_a, E_binning, v_binning)
pdf_counts_b = bin_pdf(class_b, E_binning, v_binning)
```

```
2.6315789473684212%
15.789473684210526%
28.94736842105263%
42.10526315789474%
100%
2.6315789473684212%
15.789473684210526%
28.94736842105263%
42.10526315789474%
100%
```

[142]:
```
sum(pdf_counts_a)
```

[142]:
```
0.999031
```

[143]:
```
func   = NegativeLogLikelihoodFunction(data_counts, pdf_counts_a, pdf_counts_b)
result = nlog_opt(func, num_0)
result
```

[143]:
```
 final_simplex: (array([[105.75650538, 195.85146897],
       [105.75644611, 195.85152896],
       [105.75643048, 195.85143428]]), array([-343.80856017, -343.80856017,
-343.80856017]))
           fun: -343.80856017177854
       message: 'Optimization terminated successfully.'
          nfev: 85
           nit: 42
        status: 0
       success: True
             x: array([105.75650538, 195.85146897])
```

[144]:
```
num_a, num_b, lo_a, hi_a, lo_b, hi_b = Likeli_Solve(data_counts, pdf_counts_a,␣
↪pdf_counts_b, num_0)
```

[136]:
```
print(num_a, num_b, lo_a, hi_a, lo_b, hi_b)
```

```
111.67298855337415 189.40233618534853 97.63825124790257 126.6044200673427
172.8729657522919 206.6619008565239
```

[145]:
```
print(hi_a - lo_a)
print(hi_b - lo_b)
```

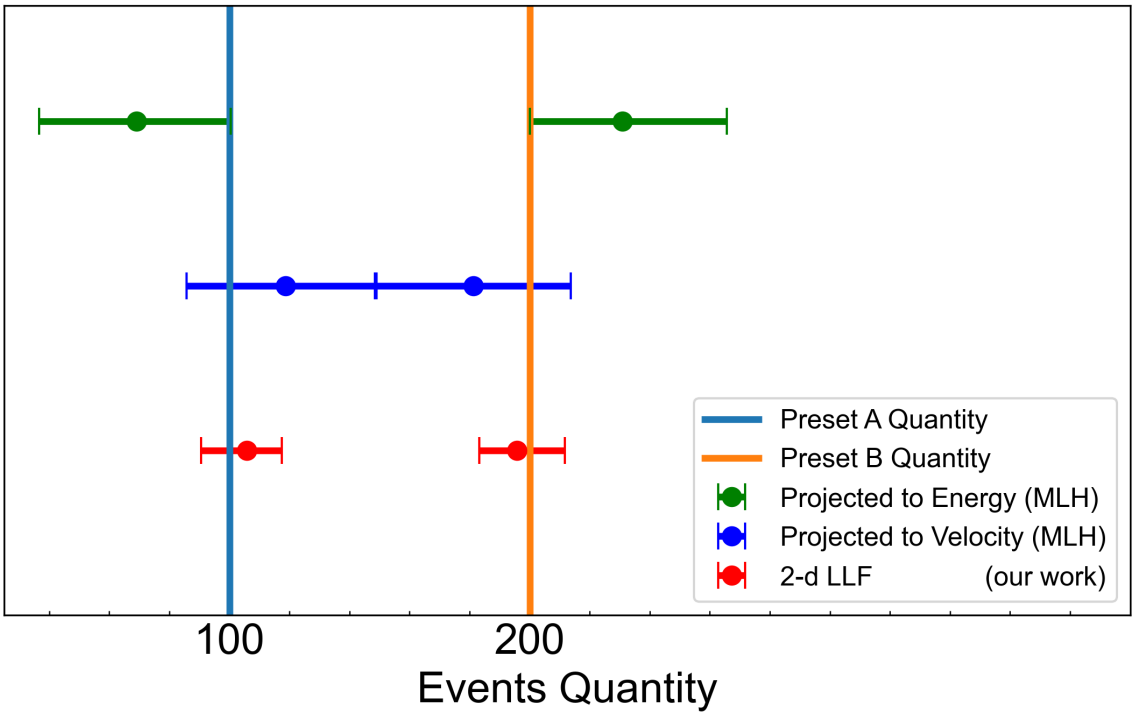
```
24.281874046012405
30.951066023883982
```

[163]:
```
print('Number of A events = $%0.2f^{+%0.2f}_{-%0.2f}$'%(num_a, hi_a-num_a,␣
↪num_a-lo_a))
print('Number of B events = $%0.2f^{+%0.2f}_{-%0.2f}$'%(num_b, hi_b-num_b,␣
↪num_b-lo_b))
```

```
Number of A events = $105.76^{+12.55}_{-11.73}$
Number of B events = $195.85^{+15.82}_{-15.13}$
```

[205]:
```
# 画一个比较图
plt.figure(figsize=(9, 5))
plt.plot([100,100],[0,4])
plt.plot([200,200],[0,4])
plt.errorbar(
    [69.07, 230.93],
    [3,3],
    xerr      = [[-31.33,+30.79],[-32.44,+34.67]],
    marker    = 'o',
    linestyle = 'none',
    color     = 'g',
    label     = 'Data',
    ms        = 8,
    elinewidth= 3,
    capsize   = 6,
)
plt.errorbar(
    [118.70, 181.30],
    [2,2],
    xerr      = [[-30.40,+32.88],[-32.88,+32.40]],
    marker    = 'o',
    linestyle = 'none',
    color     = 'b',
    label     = 'Data',
    ms        = 8,
    elinewidth= 3,
    capsize   = 6,
)
plt.errorbar(
    [105.76, 195.85],
    [1,1],
    xerr      = [[-11.73,+12.55],[-15.13,+15.82]],
    marker    = 'o',
    linestyle = 'none',
    color     = 'red',
    label     = 'Data',
    ms        = 8,
```

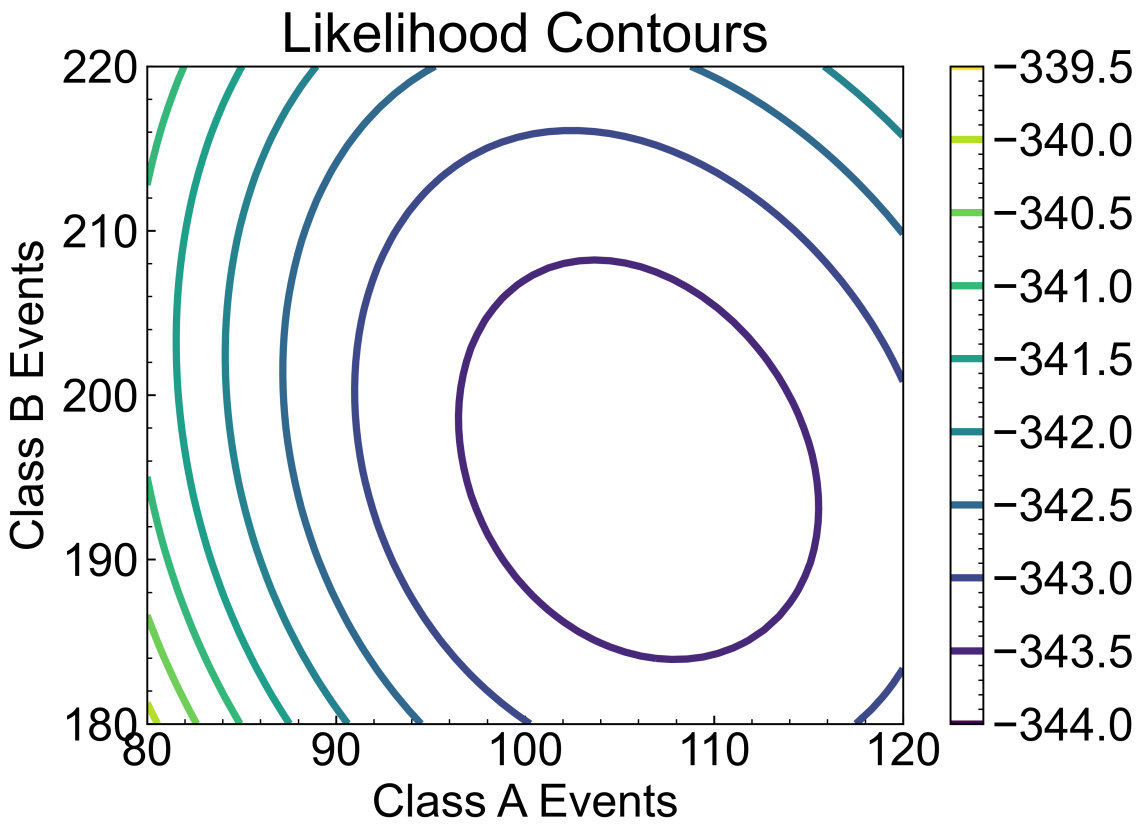
```
    elinewidth= 3,
    capsize   = 6,
)
plt.xlabel('Events Quantity')
plt.yticks([])
plt.xticks([100,200])
plt.ylim([0,3.7])
plt.legend(['Preset A Quantity', 'Preset B Quantity', 'Projected to Energy (MLH)',␣
 ↪'Projected to Velocity (MLH)', '2-d LLF                (our work)'], fontsize=12,␣
 ↪loc='lower right')
plt.xlim([25,400])
plt.savefig('img/误差棒.png')
```



[146]:
```
plt.figure(figsize=(7, 5))

X, Y = np.meshgrid( np.linspace(80,120), np.linspace(180,220) )
Z    = [ func(x,y) for x,y in zip(X.flatten(), Y.flatten()) ]
Z    = np.asarray(Z).reshape(X.shape)

plt.contour(X,Y,Z)
plt.colorbar()
plt.title('Likelihood Contours')
plt.xlabel('Class A Events')
```

```
plt.ylabel('Class B Events')
```

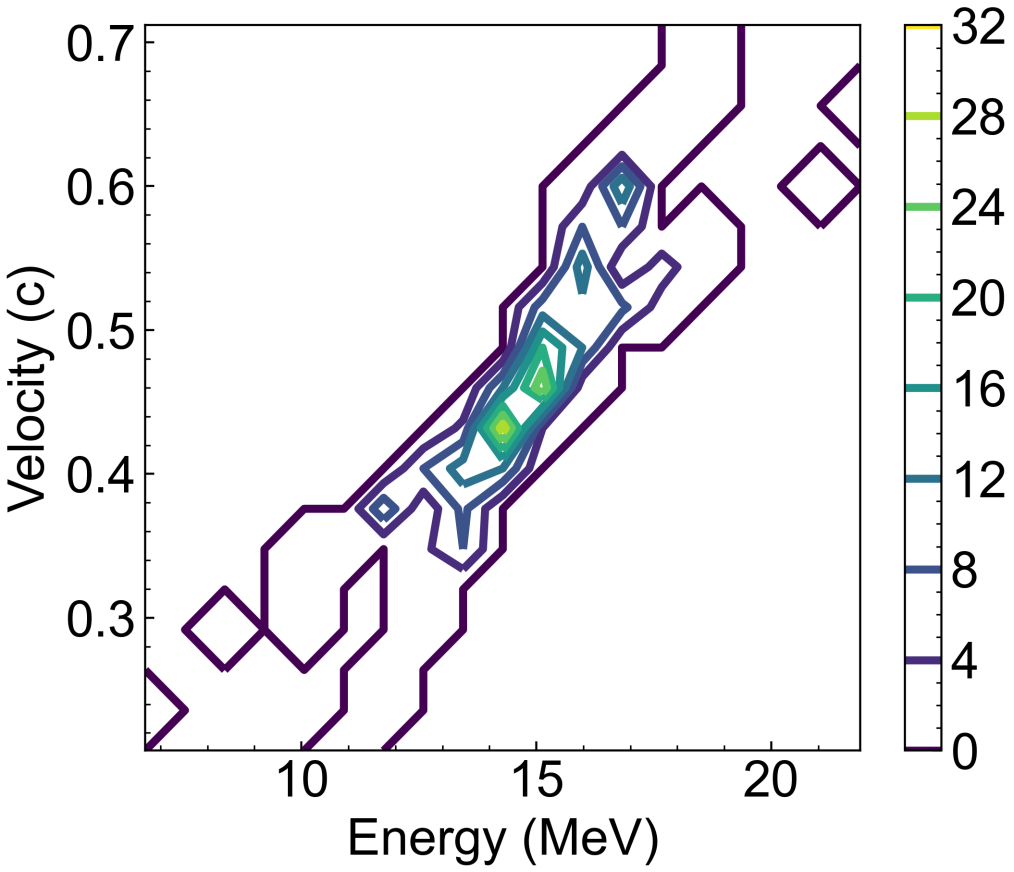
[146]: Text(0, 0.5, 'Class B Events')



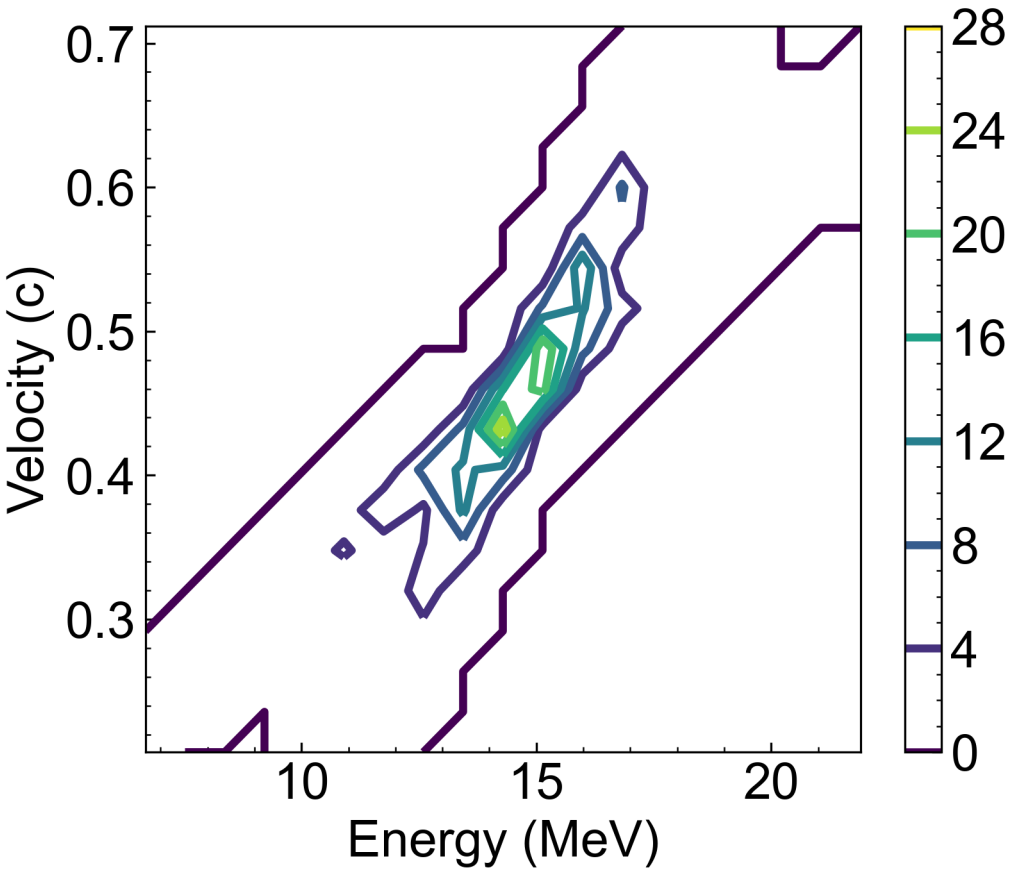
[127]:
```
# profile_class_b = profile_nlog('b', func, result.fun)
# x = np.linspace(150,215,50)
# y = [profile_class_b(nev) for nev in x]
# plt.plot(x,y)
# plt.xlabel('Class B Events')
# plt.ylabel('$\Delta$ Log ${\scr L}$')
```

[161]:
```
plt.figure(figsize=(6, 5))
plt.rcParams['savefig.dpi']          = 300
plt.rcParams['figure.dpi']           = 300       # 输出
X, Y = np.meshgrid( E_binning[:-1], v_binning[:-1] )
X    = X + (X[1]-X[0])*0.5
Y    = Y + (Y[1]-Y[0])*0.5
Z    = np.asarray(data_counts).reshape(X.shape)

plt.contour(X,Y,Z)
plt.colorbar()
```

```
plt.xlabel('Energy (MeV)')
plt.ylabel('Velocity (c)')
plt.savefig('img/直方图等高线 data.png')
```



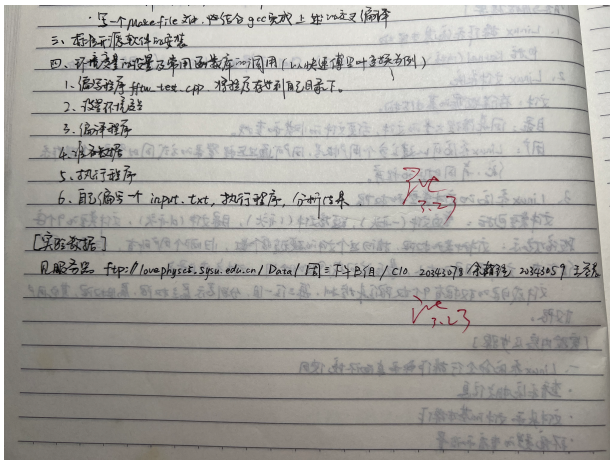
[162]:
```
plt.figure(figsize=(6, 5))
plt.rcParams['savefig.dpi']          = 300
plt.rcParams['figure.dpi']           = 300       # 输出
X, Y = np.meshgrid( E_binning[:-1], v_binning[:-1] )
X    = X + (X[1]-X[0])*0.5
Y    = Y + (Y[1]-Y[0])*0.5
Z    = np.asarray(pdf_counts_a*num_a+pdf_counts_b*num_b).reshape(X.shape)

plt.contour(X,Y,Z)
plt.colorbar()
plt.xlabel('Energy (MeV)')
plt.ylabel('Velocity (c)')
plt.savefig('img/直方图等高线 fit.png')
```

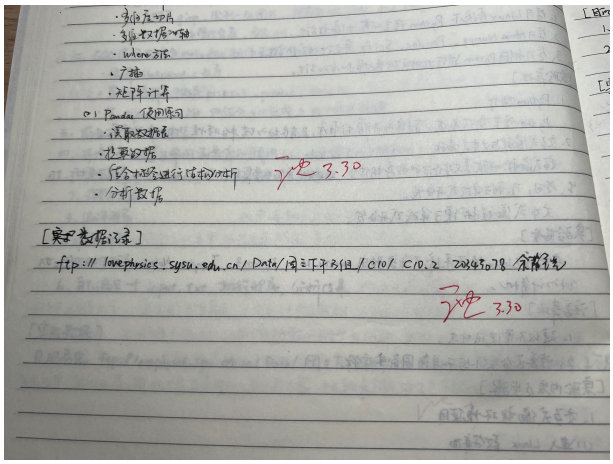


C.4 多自由度对数似然拟合

这属于拓展部分，请咨询yuyk6@mail2.sysu.edu.cn索取源代码。

D 原始数据与教师签名





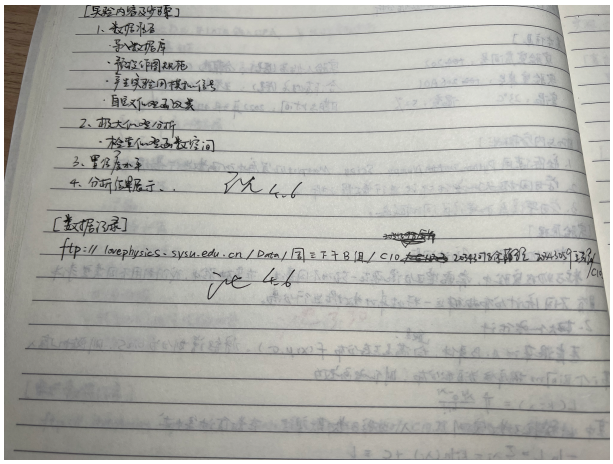


图 8 原始数据与教师签名