



生信速成（2）：基于对接的药物虚拟筛选 2

基于结构的药物设计 | 计算机辅助药物设计 | CADD

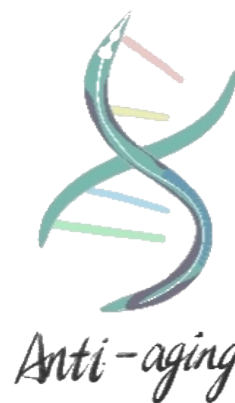
作者：Yan Pan

组织：Anti-Aging 'X' Laboratory

时间：January 30, 2024

版本：1.0

备注：仅 Anti-Aging 'X' 课题内部使用



目录

第一章 AntiAging-X-BioinformGuide 系列介绍	1
第二章 分子对接结果处理	2
2.1 分子对接模式分类	2
2.2 对接结果解读	2
2.3 分子对接结果可视化	4
2.3.1 结构导入	4
2.3.2 分子口袋可视化	6
2.4 分子对接中常见出现的作用力	7
第三章 批量分子对接	9
3.1 批量对接的概念	9
3.2 使用 Openbabel 批量处理配体结构	9
3.3 Python 基础	11
3.3.1 Python 的缩进规则	11
3.3.2 Python 代码的基本元素	11
3.3.2.1 变量	11
3.3.2.2 循环	11
3.3.2.3 函数	12
3.3.3 批量调用系统命令	12
3.4 vina 结果捕获	14
3.4.1 vina 断点续算	16
3.5 批量对接全流程	17
3.6 DEBUG	19
3.7 结语	19

第一章 AntiAging-X-BioinformGuide 系列介绍

本教程是专门为Anti-Aging'X'课题组的一年级学生设计的，旨在提供一个快速、高效的入门指南，以便他们能迅速掌握生物信息学的基础知识和技能。本教程专注于生物信息学在秀丽隐杆线虫（*C. elegans*）研究及药物筛选领域的应用，特别是在抗衰老和退行性疾病药物发现方面。此外，讲义还涵盖了iGEM（国际基因工程机器大赛）中所需的部分生物建模知识，为学生参与此类竞赛做好准备。

本教程系统地介绍了生物信息学的基本原理、主要方法和关键技术。内容包括但不限于：分子对接与动力学模拟、高通量测序数据的处理与分析、基因组信息的解读、蛋白质相互作用的网络分析、以及转录组学数据的综合应用。特别强调秀丽隐杆线虫作为模型生物在药物筛选和抗衰老研究中的重要性，详细讨论了通过生物信息学方法筛选潜在药物靶点的策略。

进一步地，通过采用案例研究和实际项目工作的模式，读者能够将理论知识应用于实际问题解决中。教程中包含了大量的实践练习和在线资源，以帮助巩固所学知识，并通过实际操作深化对生物信息学在现代生物医学研究中的应用的理解。此外，教程避免了对复杂计算过程的深入理论讨论，使得初学者，特别是生物学零基础背景的大一新生，能够快速熟悉并掌握生物信息学的基础工具和方法。通过这种方式，希望本系列教程，能成为一个引领读者深入探索生物信息学世界的桥梁。

为了更好地支持学习和资源共享，本教程的相关材料已经在 GitHub 上建立了一个专门的仓库。这个在线仓库旨在为读者提供一个动态更新的共享资源平台，其中包含了教程文档、案例研究的数据集、实践练习的代码，以及其他有用的学习资料。

- GitHub 地址：<https://github.com/Marissapy/UESTC-AntiAging-X-BioinformGuide/>

在这里你将学到：

- 极少的理论知识
- 结构生物学与物理化学的皮毛
- Python、R、Shell 的语言规范
- 基于受体结构（分子对接）的高通量虚拟筛选
- 基于配体结构（药效团等）的高通量虚拟筛选
- 蛋白质结构的预测：AlphaFold2 的使用
- 分子对接与分子动力学模拟（MD）
- 机器学习与化合物编码
- 药代动力学——小分子的 ADMET 评估
- 组学分析

更希望你学到：

- 科研入门的基本能力
- 独立思考
- 尝试独立解决问题

每个伟大的科学探索都始于一个简单的好奇心，即将开始的生物信息学之旅，不仅是对知识的探索，更是自我发现和成长的旅程。

第二章 分子对接结果处理

2.1 分子对接模式分类

分子对接通常分为三种类型:

刚性对接 (Rigid Docking) :

在刚性对接中，受体和配体都被视为刚性结构。这意味着在对接过程中，它们的形状和构象保持不变。这种方法计算简单、速度快，但不能很好地模拟真实生物体内的环境，因为在现实中，很多分子（尤其是大分子如蛋白质）在与其他分子结合时，其构象会发生改变。

半柔性对接 (Semi-flexible Docking) :

在半柔性对接中，通常将配体视为柔性的，而受体则视为刚性的。这种方法认为，配体（通常较小）在与受体（通常较大）结合时，可以改变其构象（例如，旋转键的旋转），而受体的结构则保持不变。这种类型的对接相对于刚性对接更加真实，因为它允许配体根据受体的结构进行一定程度的适应性变化。Autodock Vina 采取半柔性对接。这意味着，Vina 能够考虑配体的多种构象，使其能够探索不同的可能性，以找到最佳的对接姿态。这是通过在对接过程中允许配体的旋转键自由旋转来实现的。然而，由于受体被视为刚性的，Vina 并不考虑受体在对接过程中的构象变化。

全柔性对接（Flexible Docking）：

在全柔性对接中，受体和配体都被视为柔性的。这意味着在对接过程中，它们都可以改变构象。这种方法是最复杂也最耗时的，但它能提供最接近**真实情况的模拟**，因为在真实生物体内，许多分子在相互作用时都会发生构象的变化。常常通过半柔性对接的方法得到初步构象，再通过分子动力学模拟（全柔性对接）进行进一步验证。

2.2 对接结果解读

绪接上文，完成一轮对接后，在本地会输出 **output.pdbqt**，同时在控制台会输出如下结果：

```

AutoDock Vina v1.2.3
#####
# If you used AutoDock Vina in your work, please cite: #
#
# J. Eberhardt, D. Santos-Martins, A. F. Tillack, and S. Forli #
# AutoDock Vina 1.2.0: New Docking Methods, Expanded Force #
# Field, and Python Bindings, J. Chem. Inf. Model. (2021) #
# DOI 10.1021/acs.jcim.1c00203 #
#
# O. Trott, A. J. Olson, #
# AutoDock Vina: improving the speed and accuracy of docking #
# with a new scoring function, efficient optimization and #
# multithreading, J. Comp. Chem. (2010) #
# DOI 10.1002/jcc.21334 #
#
# Please see https://github.com/ccsb-scripps/AutoDock-Vina for #
# more information. #
#####
Scoring function : vina

```



```

Rigid receptor: /receptor.pdbqt
Ligand: /ligand.pdbqt
Grid center: X -6.67 Y 19.56 Z 1.75
Grid size : X 15.55 Y 11.14 Z 11.97
Grid space : 0.375
Exhaustiveness: 8
CPU: 2
Verbosity: 1

Computing Vina grid ... done.
Performing docking (random seed: -825848520) ...
0% 10 20 30 40 50 60 70 80 90 100%
|----|----|----|----|----|----|----|----|----|----|
*****

mode | affinity | dist from best mode
| (kcal/mol) | rmsd l.b. | rmsd u.b.
-----+-----+-----+-----
1      -5.948      0      0
2      -5.845      8.119     10.02
3      -5.833      1.151      3.442
4      -5.676      7.135      8.685
5      -5.459      2.383      3.448
6      -5.357      5.049      6.958
7      -5.292      1.655      2.128
8      -5.285      8.549     10.41
9      -3.674      8.213     10.4

```

在参数部分: **Scoring function**: 使用的评分函数, 这里是 Vina; **Rigid receptor**: 指定了静态受体的文件位置 (/receptor.pdbqt); **Ligand**: 指定了配体的文件位置 (/ligand.pdbqt); **Grid center** 和 **Grid size**: 定义了对接搜索的空间区域; **Exhaustiveness**: 表明了对接搜索的彻底程度, 数值越高, 搜索越详尽; **CPU**: 使用的 CPU 核心数量。

在对接结果部分, **mode**: 表示不同的对接模式或姿态。 **affinity (kcal/mol)**: 表示预测的亲合力, 用千卡/摩尔表示。数值越低, 表示配体与受体的结合越紧密, 亲合力越强。 **rmsd l.b.** 和 **rmsd u.b.**: 分别表示低界和上界的均方根偏差 (RMSD), 用于衡量不同模式之间的结构差异。RMSD 值越小, 表示结构差异越小。

在这个具体的输出结果中, 可以看到共有 9 种不同的对接模式。**mode 1** 具有最低的亲合力 (**-5.948 kcal/mol**), 通常认为亲合力最强的模式是最可能的对接姿态。其他模式的亲合力相对较高, 但它们提供了可能的替代对接方式。RMSD 值提供了这些模式相对于最佳模式的结构差异信息。

定义 2.1 (什么是 RMSD?)

RMSD 在分子模拟领域, 尤其是动力学模拟中是一个重要的概念。RMSD 即 **均方根偏差 (Root-Mean-Square Deviation)**, 是一种衡量分子结构差异的数学量度。在分子对接和结构生物学中, RMSD 用于量化两个分子结构 (例如, 蛋白质或配体的不同构象) 之间的相似性或差异。计算 RMSD 时, 通常会比较分子中相应原子的位置。

想象一下, 你有两套相同的乐高模型, 它们由许多不同颜色和形状的积木组成。现在, 你把这两套乐高模型各自拼装成一个相同的结构, 比如说一座小房子。但是, 由于每次拼装都可能略有不同, 所以这两座小房子在一些细节上可能会有些微的差别。

RMSD 就像是一个工具, 帮助你测量这两座乐高房子在细节上的差异有多大。你会检查每一块对应位置

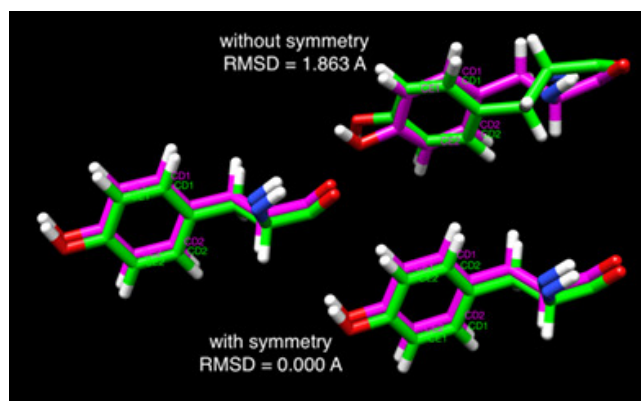


图 2.1: RMSD 的结构计算示例。

的积木，在两座房子中是否完全一样放置，或者有多大的差距。然后，RMSD 会计算出一个数值，这个数值越小，说明两座房子在细节上越相似；数值越大，说明它们之间的差异越明显。

在分子对接的情境中，RMSD 帮助人们理解当他们尝试将一个小分子（比如药物）与一个大分子（比如蛋白质）结合时，这个小分子可能采取的不同摆放位置有多相似。如果 RMSD 值很小，意味着小分子在不同模拟中占据了非常相似的位置和姿态；如果 RMSD 值很大，则表示小分子的位置和姿态在不同模拟中有较大差异。

Vina 计算 RMSD 的方法遵循标准的 RMSD 计算步骤，但专门针对配体的对接姿态（因为 vina 采取半柔性对接）。

首先，选取配体分子的每个原子，并找到其在不同对接姿态中的对应原子。然后，测量这些对应原子在三维空间中的位置差异。接下来，计算每对原子距离差的平方，然后将所有这些平方值求和。最后，将这个总和除以原子的总数，得到平均平方差，然后对这个平均值取平方根，得到 RMSD 值。

在 Vina 的输出中，通常会列出每个对接姿态相对于最优姿态（即亲和力和最高的姿态）的 RMSD 值。这个数值越小，表示该姿态与最优姿态在结构上越相似；数值越大，表示结构差异越显著。

2.3 分子对接结果可视化

2.3.1 结构导入

Autodock Vina 采取半柔性对接。这意味着，由于受体被视为刚性的，Vina 并不考虑受体在对接过程中的构象变化。所以，在输出文件 output.pdbqt 中，仅输出了对接时配体的结构。

在此，仅展示最简单的可视化和分析教程，更高级的方法可以参考 B 站的教程。首先，从服务器上下载对接结果（output.pdbqt）。

在 pyMol 中打开用于分子对接的受体结构（pdb 或 pdbqt 格式均可），再从 **File-Open** 导入（或直接拖动输出文件到 pyMol 中）输出的 output.pdbqt，如图 2.2。导入后，可依次点击右侧的 **A-rename object** 为结构改名以便后续操作。在示例中，mTOR 蛋白与小分子 Aspirin 分别被改为了 **receptor**（受体）与 **ligand**（配体）。为了方便观察，打开 **Display-Sequence** 的序列窗口，并将蛋白质按照链号标色（点击 **C-by Chain**）。

在右侧小分子的名称旁会出现 1/9，为 vina 输出的 9 个不同的配体构象。在右下角的箭头中可以依次观察不同的构象，如 2.3。在对接时因为保留了不同的 AB 链，可以观察到小分子实际上与 AB 链都可发生对接。在不同的构象中，小分子与 AB 链的距离也有所变化。实际上，分子对接本质是一种预测，我们并不能得知 Aspirin 与 mTOR 这两种完全毫无相干的物质能否发生结合，也不能确定是与哪个链发生结合，分子对接仅仅为我们提供了一种可能性。在这个问题中，需要分子动力学模拟（Molecular Dynamics, MD）的进一步验证，动力学模拟可以为研究人员提供体系内所有原子的动态变化过程以及物理量，包括亲合能。

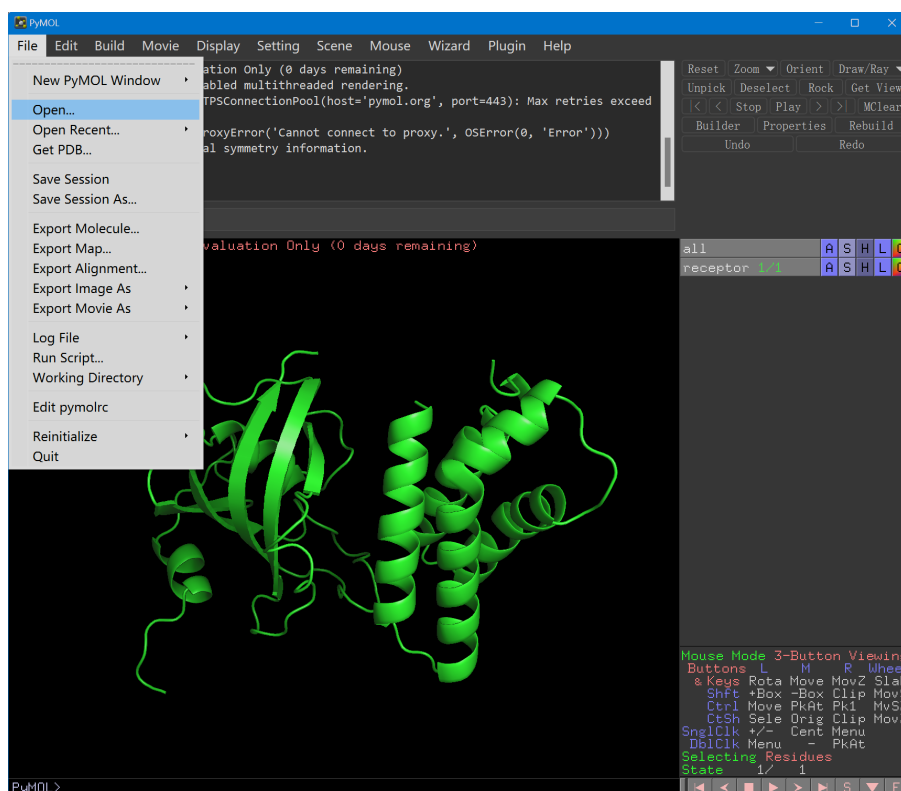


图 2.2: 在 pyMol 中打开处理后的受体结构，再导入输出的配体。

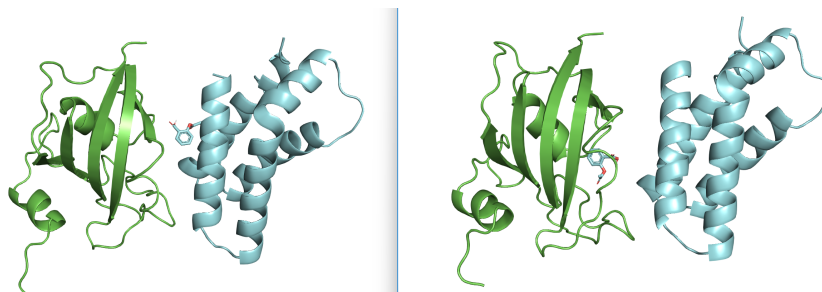


图 2.3: Aspirin 的构象 1 与构象 2 与 mTOR 上不同的链对接。

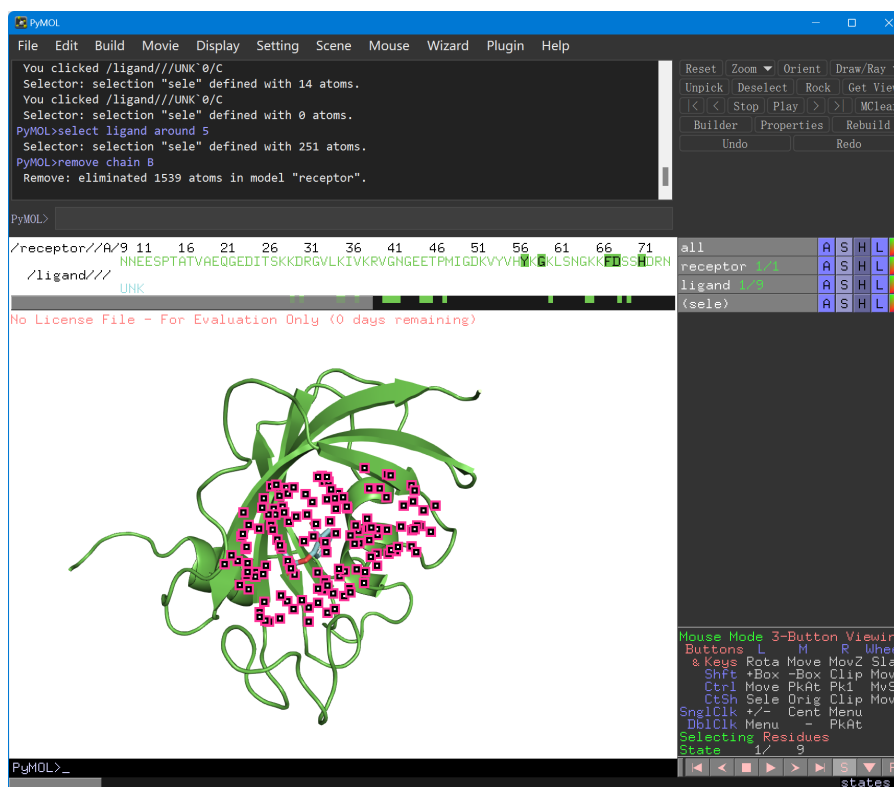


图 2.4: 选择配体周围 7 埃的所有原子，作为分子口袋。

2.3.2 分子口袋可视化

以构象 1 为例。首先更换背景颜色及画质。点击 **Display-Background-White** 调整为白色背景方便展示，再点击 **Display-Quality-Maximum quality** 选择最高画质。选择 **Display-Color space-CMYK** 调整颜色（可选）。
笔记 CMYK 与 RGB 色彩模式不同，后者是用于电子显示设备（如电脑屏幕、电视和手机）的增色法（additive color method），通过叠加红色、绿色和蓝色光来生成颜色。相比之下，CMYK 更适用于反映光的物质（如纸张），而 RGB 更适用于发射光的物质。在印刷行业中，CMYK 是标准色彩模式。

在构象 1 中，配体离 A 链距离较近（在 Sequence 窗口可确定对应的链号），其分子口袋（即结合位点）的构成与 B 链无关，故先删除 B 链。删除 B 链后，选择配体周围 7 Å 的原子，在 pyMol 的控制台输入如下代码所示。

```
# 删除B链
remove chain B
# 选择配体周围7埃的所有原子，select命令后跟的是结构名称，需要与你的小分子名（见pymol右侧）相同。若执行后
发现包围的面积过大和过小，可适当调整至美观的距离。
select ligand around 7
```

执行完 select ligand around 5 后，会选择上一部分氨基酸，此时，右侧会出现（sele），即为刚才选择的分子口袋。点击 **A-Extract object** 后，会出现一个单独的结构“obj01”，即为分离后的分子口袋。对 obj01 点击 **A-generate-vaccum electrostatics-protein contact potential(local)**。正常情况下，会出现分子口袋与配体的界面。若未出现，注意是否有报错，有可能因为在 select ligand around 7 一步范围过小，退出软件重试。

此时，还可以更改原蛋白质或配体的配色以便观看，如图 2.5，展示了阿司匹林（Aspirin）分子与哺乳动物雷帕霉素靶蛋白（mammalian target of rapamycin，简称 mTOR）A 链的结合口袋的分子对接示意图。在图中，阿司匹林分子通过分子对接技术被放置于 mTOR 蛋白的活性口袋中。蛋白质的三维结构用灰色的带状图表示，而口袋区域则通过色彩编码的表面来标示，显示了分子口袋内部的电荷分布。负电荷区域用红色表示，正电荷区域用蓝色表示，这有助于理解分子间的电荷互补性。图中的颜色条显示了电荷分布的范围，从 -76.871（红色）到 +76.871（蓝色），单位未指明但通常为 kilojoules per mole (kJ/mol) 或 kilocalories per mole (kcal/mol)，1KJ=4.184Kcal。

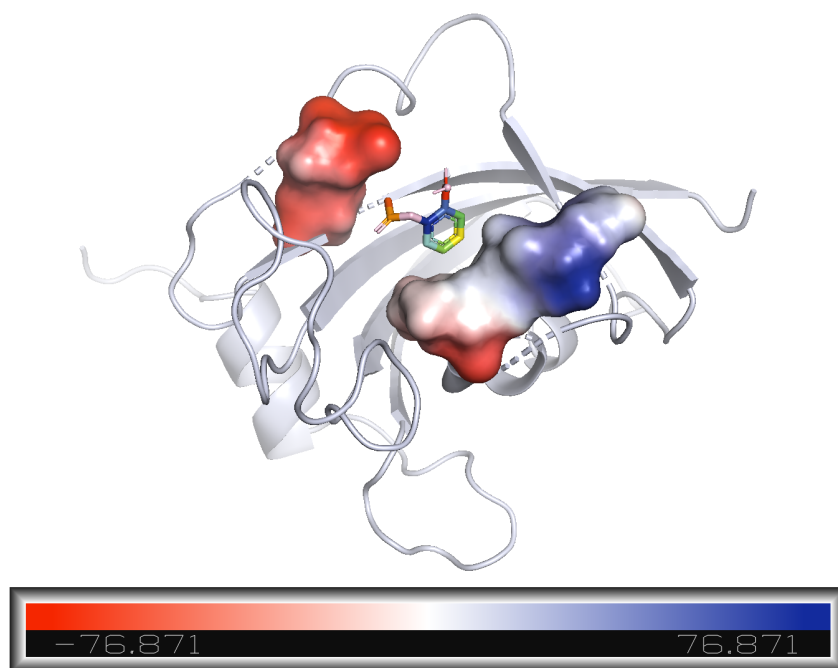


图 2.5: 调整好的分子口袋示意图。

定义 2.2 (什么是分子口袋?)

分子口袋是分子结构中的一个区域，通常是蛋白质或其他大分子中的凹陷或空腔。这些口袋在生物化学和药物设计中非常重要，因为它们允许小分子（如药物、底物或配体）特异性地结合到大分子上。分子口袋的特性（如大小、形状、化学性质和电荷分布）决定了它们与哪些小分子结合，以及结合的紧密程度。在药物设计中，研究者通常寻找或设计小分子，这些小分子可以精确地适配到目标蛋白质的分子口袋中。这种适配可以抑制或激活蛋白质的功能，从而对疾病过程产生治疗作用。例如，在开发针对酶或受体的抑制剂时，研究者会努力设计一个能够适合并阻塞这些蛋白质活性部位的分子。



结构可视化抛砖引玉仅介绍到此，关于分子对接还有氢键、 π - π 堆叠、疏水作用力等作用可视化，最简单的方式是在 CB-Dock2(<https://cadd.labshare.cn/cb-dock2/php/blinddock.php>)中进行网页对接，之后会输出优美的可视化结果，足够应付大部分需求。至于更深入的可视化方法待自行去 B 站学习，非常简单。

至于分子互作，可以在 pyMol 右侧对配体点击 **A-Find**，即可勾选你需要的作用力。若勾选完没反应（配体与受体之间没有出现虚线），即为不存在。

2.4 分子对接中常见出现的作用力

氢键：氢键是一种较弱的化学键，发生在带有部分正电的氢原子和另一带有部分负电的电负性原子（如氧、氮）之间。在分子对接中，氢键对分子间的特异性识别非常重要。例如，在药物与其靶标（通常是蛋白质）之间的相互作用中，氢键可以增加两者的亲和力和特异性。

π - π 堆叠 (π 堆叠)： π - π 堆叠是一种发生在芳香环之间的相互作用，特别是在这些环平行排列时。这种作用力在蛋白质的三级结构中以及蛋白质与多环芳香药物之间的相互作用中扮演重要角色。

疏水作用力：疏水作用力或疏水相互作用指的是非极性分子或分子的非极性部分倾向于相互靠近，从而避免与水分子接触的现象。在分子对接中，疏水作用力有助于稳定蛋白质的内部核心以及蛋白质与非极性药物分

子之间的结合。

离子作用：离子作用是指带相反电荷的离子或分子部分之间的吸引作用。这种作用力在电荷互补的分子（如酸性药物与碱性残基之间）的结合过程中非常重要，通常在分子对接的过程中提供强大的结合能。

第三章 批量分子对接

3.1 批量对接的概念

批量分子对接常常指多个小分子（几千、几万个或更多）在一个蛋白质受体的分子口袋内进行对接的过程，旨在通过高效、高通量的方式筛选出最优亲合能的小分子，即最有可能结合，成为酶或受体的抑制剂的小分子。

在前文中，我们了解了单个分子对接的全过程：处理受体结构，寻找活性中心（即配置 `config.txt` 文件），处理配体结构，运行对接程序。而批量值的是单个受体对多个配体，即我们仍需要手动处理受体结构和配置 `config.txt`，然后再批量处理配体结构，最后进行批量对接。

3.2 使用 Openbabel 批量处理配体结构

在此，我准备好了一个商用中药小分子数据库，并已经将其转换成单个的 3D 结构 `pdb` 文件，可以在 Github 上下载，文件名为 `Structures_Chinese_Medicine.zip`，<https://github.com/Marissapy/UESTC-AntiAging-X-BioinformGuide>。下载后上传到服务器的文件夹上。

定义 3.1 (小分子数据库如何获取?)

一般来说，TCMSP（中药小分子数据库）、Pubchem 数据库都只能下载单个的小分子结构。ZINC 数据库等分子过多，筛选太慢，很可能筛出来的小分子还无法购买。这种情况下，一般都选用商业的化合物数据库（常常会有销售来实验室推销）。

但拿到手的数据库常为单个的 `sdf` 文件，里面集成了几千个化合物的 2D 结构，而分子对接需要单个的 3D 结构，此时即需要将原文件拆分和转换 3D。在转换 3D 过程中，因为出现手性异构体等，会非常占用时间，所以需要耐心等待。

Openbabel 软件常用于完成上述操作，它是处理化学结构（尤其是小分子）的一大帮手。在控制台中，仅需要加上 `-gen3d` 参数，即可将 2D 的小分子转换为 3D。

下载好 `Structures_Chinese_Medicine.zip` 后，在服务器新建一个文件夹（假设为 `dock02`），将压缩包移动进文件夹内。如果是通过 RStudio 上传的 `zip` 文件，即会自动解压。所以不要直接把压缩包传到 `home` 目录！两千多个文件会全占满的，非常不好操作。

若没有自动解压，则需要使用 `unzip name.zip` 命令（需要修改文件名），`unzip` 命令可使文件解压到本地。此时，在文件夹内，会出现两千多个小分子的 `pdb` 结构，需要使用 Openbabel 工具将其统一转换为 `pdbqt`。

Openbabel 的用法具体参照文档：https://open-babel.readthedocs.io/en/latest/Command-line_tools/babel.html，非常强大，本章仅作粗略阐释。Openbabel 在服务器已经安装，可以直接使用，在私人电脑的 WSL2 上，需要下面的命令进行安装：

```
sudo apt-get install openbabel
```

使用命令行，将工作目录切换到刚才新建的文件夹下，使用 `ls` 确保本目录下所有的 `pdb` 文件。

```
cd dock02
ls
```

使用 `openbabel` 命令，将所有的 `pdb` 文件转化为 `pdbqt` 文件。

```
obabel -ipdb *.pdb -opdbqt -m
```

在服务器上，需要等待几分钟。在家用电脑上，需要 48 小时以上的时间。在这几分钟内，可以了解一下 openbabel 的用法：

基本语法:

```
obabel [-i <input-ID>] infile [-o <output-ID>] [-O outfile] [OPTIONS]
```

在命令行中, **-i** 常常表示”input”。**obabel -ipdb** 表示使用 openbabel, 输入的文件为 pdb 格式。如果需要输入 sdf 格式的文件, 即改为 **obabel -isdf**, 其他格式也类似。在 **-ipdb** 后需要紧跟输入文件的名称, 在绝大部分命令行环境中, 星号 * 都意味着所有。即 *.pdb 意味着选中本目录下所有的 pdb 文件。类似地, **-o** 代表”output”, **-opdbqt** 意味着需要输出 pdbqt 格式的结构。查询官网文档得知, **-m** 意为输出多个文件, 用于批量转换。

最后, 会提示:

```
9126 molecules converted
5608 files output. The first is Structures__11-Deoxymogroside_IIE_cdx.pdbqt
```

到此, 所有的小分子都已经被转换完成。再次使用 ls 查看当前目录下的所有文件, 会发现多了很多 pdbqt 格式的结构文件。在使用 vina 做分子对接的时候, 在 config.txt 中里有配体参数”--ligand”, 后跟的是配体结构文件的名称。在此, 我们只要不断地更换 ligand 参数后跟的配体名, 把当前目录下的所有配体结构都遍历一遍, 即可达到效果。

这里需要做一点小小的改动。在 config.txt 中, 需要删除 ligand 参数的一行 (“--ligand=ligand.pdbqt”), 也需要删除 out 参数。删除后如下:

```
receptor = protein.pdbqt
ligand = C01.pdbqt
center_x = 0
center_y = 0
center_z = 0
size_x = 16
size_y = 16
size_z = 16
exhaustiveness = 30
num_modes = 5
```

当然, config.txt 中关于受体的信息 (比如对接 box 的中心坐标、大小) 需要根据不同的受体来确认。删除掉 ligand 参数是因为我们需要一直更换配体名称, 仅仅想保留固定的参数在 config.txt 中, 删除掉 out 参数是因为批量对接只需要获得亲合能, 不需要输出对接的结构, 不然就会占用过多。但 ligand 参数是必不可少的, 删除后, 我们运行 vina 的方式改为如下:

```
vina --ligand=ligand_1.pdbqt --config=config.txt
```

假设当前目录有 ligand_1/2/3.pdbqt 等文件, 则批量对接要做的就是改变 ligand 参数, 逐个遍历, 如下:

```
vina --ligand=ligand_1.pdbqt --config=config.txt
vina --ligand=ligand_2.pdbqt --config=config.txt
vina --ligand=ligand_3.pdbqt --config=config.txt
vina --ligand=ligand_4.pdbqt --config=config.txt
```

但人工是不可能逐个执行这么多的命令的。这时, 需要脚本来帮助我们。在逻辑上, 首先我们需要读取当前目录下的所有 pdbqt 格式的文件 (即小分子配体), 将它们的文件名存储入列表内。再依次执行”vina --ligand=ligand_1.pdbqt --config=config.txt”命令, 只需要在循环中替换掉配体名称即可。

在对接前, 请保证你处理好的受体、配体的 pdbqt 文件以及 config.txt 在同一目录下。

3.3 Python 基础

本章仅用最简单的语言来编写 **vina** 批量对接的所有 **Python** 代码，仅作最简解释，系统性学习 **Python** 可以多看看其他教程，对 **python** 不感兴趣也可以直接跳过。

定义 3.2 (什么是 Python?)

Python 是一种流行的编程语言，被广泛用于各种领域，如网站开发、数据分析、人工智能等。它以简洁易读而著称，非常适合初学者学习。

3.3.1 Python 的缩进规则

在 Python 中，缩进用来表示一段代码属于特定的代码块。这与其他许多编程语言不同，后者通常使用花括号来定义代码块。

```
def example_function():
    # 这是一个函数内的代码块
    print("Function start")
    for i in range(3):
        # 这是一个循环内的代码块
        print(i)
    # 这回到了最外层的代码块
    print("Function end")
```

推荐使用 4 个空格进行缩进。

3.3.2 Python 代码的基本元素

3.3.2.1 变量

在编程中，变量是用于存储数据的基本单位。你可以把变量想象成一个存储箱，你可以放入一些数据，稍后再取出来使用。在 Python 中，变量的创建非常简单：首先是**声明变量**，你只需写下变量名和你想赋予它的值。例如，**number = 10**。这里，**number** 是变量名，而 10 是它存储的值。

Python 是一种动态类型的语言，这意味着你不需要提前声明变量的类型。变量可以存储不同类型的数据，如整数 (int)、浮点数 (float)、字符串 (str) 等。

一旦变量被赋值，你就可以在代码的其他部分使用它。例如，**print(number)** 将会在屏幕上显示变量 **number** 的值。

3.3.2.2 循环

循环是编程中用于重复执行一段代码的结构。在 Python 中，最常见的循环类型是**for**循环和**while**循环。

： **for** 循环用于遍历一系列项，是最常用的循环语句。这些项可以是列表中的元素、字符串中的字符或任何可迭代的序列。

在 Python 中，**range** 是一个非常有用的内置函数，主要用于生成一个**数字序列**。这个序列通常用在循环中，特别是 **for** 循环，来迭代固定次数。**range** 函数非常灵活，可以根据需要生成不同的数字序列。下面是 **range** 与 **for** 的一些基本用法和特点：例如：

```
for i in range(5):
    print(i)
# 单参数range，这段代码会打印从0到4的数字。
```

```
for i in range(2,5):
    print(i)
# 双参数range, 为左闭右开的区间, 这段代码会打印"2","3","4"。
for i in range(2, 10, 2):
    print(i)
# 三参数range, 常用于等差数列的生成, 前两个参数左开右闭, 第三个参数为步长(公差)。会生成数字2, 4, 6, 8。
```

在使用 for 循环时, 注意考虑缩进。

而 while 循环会一直执行, 直到一个指定的条件不再满足。例如:

```
i = 0
while i < 5:
    print(i)
    i += 1 # 这一步相当于 i = i+1, 即 i 自增。
```

设定初始的 i 为 0, 进入循环后, 只要满足 $i < 5$ 就一直执行, 每次的循环到最后一步时, i 自增 1。直到 i 增加到 5 停止循环。

3.3.2.3 函数

函数是一段组织好的, 可重复使用的, 用来实现单一或相关联功能的代码块。函数可以提高代码的模块化和代码重用率。在之后的编写中, 会遇到函数的调用。

3.3.3 批量调用系统命令

我们需要使用 Python 来读取当前目录下的所有 .pdbqt 文件, 并将它们分别带入到 AutoDock Vina 命令中作为 -ligand 参数。想必大家都听说过“调包侠”一词, 用于形容(特别是 Python)工作者疯狂调用别人写好的模块来完成工作, 这便是 python 有魅力的特点。

模块是 Python 中包含预先编写好的代码(如函数、变量等)的文件。我们使用 import 语句来导入模块。例如, “import glob”导入了“glob”模块, 该模块可以用于查找匹配特定模式的所有路径名。在此, 我们使用 glob 模块来查找路径名。

在工作目录下(与受体、配体、config.txt 在同一目录), 新建一个 python 文件, python 脚本的后缀名往往是 .py。

```
nano mutiple-dock.py
```

nano 用于在 Linux 的控制台内进行文本编辑, 当文件不存在的时候, 会默认新建。让我们新建一个 mutiple-dock.py。或者, 你可以在私人电脑的编辑器(Jupyter Lab、pyCharm、VS Code 等)甚至在服务器的 RStudio 上进行编辑。

打开空白的 python 脚本后, 第一件事就是调包。我们需要导入“glob”模块以获取本地目录的所有结构文件。调包使用“import”。

定义 3.3 (Python 中的包管理: pip)

pip 是 Python 的官方包管理器, 它是一个命令行程序, 用于安装、升级和管理 Python 包。Python 包是包含 Python 代码的目录, 这些代码通常是为了实现特定的功能而编写的。pip 从 Python 包索引(Python Package Index, 简称 PyPI)获取包, PyPI 是一个存放 Python 软件包的在线仓库。

使用 pip 安装模块的基本命令格式是: **pip install package_name**, 这里 package_name 是你想要安装包的名称。例如, 要安装一个流行的科学计算包 NumPy, 你可以执行: **pip install numpy**。

所以, 在你尝试调用包时, 若遇到报错告诉你无此模块, 请先使用 pip install 来安装。

```
import glob
# 获取当前目录下的所有pdbqt文件
pdbqt_files = glob.glob('*.pdbqt')
```

我们使用 `glob.glob('*.pdbqt')` 查找所有以 `.pdbqt` 结尾的文件。在这里的 `glob.glob` 中，前一个 `glob` 表示需要调用 `glob` 模块，后一个 `glob` 表示需要调用本模块的 `glob` 函数。`*` 表示任何字符的序列，`.pdbqt` 是文件扩展名，`*.pdbqt` 表示匹配当前目录下的所有 `pdbqt` 文件。读代码时，需要从右往左看。即使用 `glob` 读取完后，将所有匹配到的文件名都存入一个叫 `pdbqt_files` 的变量中。

读取完所有的文件名后，这时仅需要逐个替换掉 `vina` 的 `ligand` 参数进行逐个对接。让我们先定义 `vina` 命令的主体部分，也当作字符串存入变量中，在我们需要执行这个命令时，只需要找到这个变量。同时，变化的部分用“`{}`”来代替，这时占位符的意思。这一点，我们之后会细讲。实际上是使用了“字符串格式化”的工具，字符串格式化是一种在字符串中插入或替换某些片段的技术，可以将变量插入到一个预定义的字符串模板中。在你的代码中继续写：

```
# Vina命令的基本部分
vina_command_base = "vina --ligand={} --config=config.txt"
```

现在，我们只需要一个循环，来遍历 `pdbqt_files` 变量中的所有元素（所有配体的文件名），只需要用到 `for` 循环来遍历。“`for file in pdbqt_files`”的意思是，在 `pdbqt_files` 变量中从头逐个取出元素，在每次循环迭代中，临时变量 `file` 都会被赋予列表中的下一个文件名。

举个例子，如果 `pdbqt_files` 列表包含了 `['file1.pdbqt', 'file2.pdbqt', 'file3.pdbqt']`，那么循环将依次处理这三个文件。在第一次迭代中，`file` 将等于 `'file1.pdbqt'`，在第二次迭代中，`file` 将等于 `'file2.pdbqt'`，以此类推。

前文提到字符串格式化，现在，在 `for` 循环中，已经取到了小分子的文件名，即可构建完整的 `vina` 命令了。“`vina_command_base.format(file)`”的意思是，使字符串变量“`vina_command_base`”格式化，并向占位符内填充（替换“`{}`”）“`file`”变量。完成的 `vina` 命令将赋值给一个新的变量“`vina_command`”。

这时，只需要在你的系统中运行完整的 `vina` 命令即可。但 `vina` 需要在 Linux 的控制台内执行，python 如何执行 Linux 系统层面的工作？需要调用 `os` 模块，使用它的“`system`”函数。当你调用 `os.system` 并传递一个命令字符串时，该命令会在系统的命令行或终端中执行。

先在文件开头调用“`os`”模块，再在你的代码中输入：

```
for file in pdbqt_files:
    # 构建完整的Vina命令
    vina_command = vina_command_base.format(file)

    # 运行Vina命令
    os.system(vina_command)
```

完整的脚本，如下所示：

```
import glob
import os

# 获取当前目录下的所有pdbqt文件
pdbqt_files = glob.glob('*.pdbqt')

# Vina命令的基本部分
vina_command_base = "vina --ligand={} --config=config.txt"

for file in pdbqt_files:
```

```
# 构建完整的Vina命令
vina_command = vina_command_base.format(file)

# 运行Vina命令
os.system(vina_command)
```

在此，如果你尝试运行脚本，vina 便会开始批量对接。但 vina 输出的亲合能结果仅限于控制台观察，我们如何得到批量对接后的结果？这时，我们就需要对 vina 在控制台输出的结果进行保存。

3.4 vina 结果捕获

在本节，请先看代码意会，再看文字解释，不要把自己看晕了，本节可跳过。

我们需要将 vina 执行每一次对接后输出的结果以文本的方式保存下来。假如我希望 vina 将所有结果保存到工作目录的/results 文件夹内。需要首先确保这个文件夹存在，如果不存在，则创建它。

在此，又到了 os 包发挥的时间，我们可以使用 `os.path.exists` 检查 results 文件夹是否存在，如果不存在，则使用 `os.makedirs` 创建它。这样的条件语句如何书写呢？python 的条件语句非常简单。

```
# 确保结果文件夹存在
result_folder = 'results'
if not os.path.exists(results_folder):
    os.makedirs(results_folder)
```

为了捕获 vina 的输出，我们选择放弃 os 包来执行命令行，对于更复杂和安全性更高的任务，通常建议使用 subprocess 模块。subprocess.run 是 Python 中用于执行外部命令的函数。它是 subprocess 模块的一部分，提供了比 os.system 更多的控制能力，包括捕获命令的输出、错误处理等。

我们使用“`subprocess.run(vina_command.split(), capture_output=True, text=True)`”来执行 vina 命令。这里的 `vina_command.split()` 中的 `split()` 为习惯性用法，意思是字符串分割成一个列表，列表中的每个元素都是命令行中的一个部分。例如，如果 `vina_command` 是“`vina -ligand=file.pdbqt -config=config.txt`”，`vina_command.split()` 会将其分割为“`[\"vina\", \"-ligand=file.pdbqt\", \"-config=config.txt\"]`”，好处在于方便提供参数列表，作为习惯性用法，不用深究。

我们还需要 `capture_output` 这个参数，并把它改为 `True`，这个参数告诉 `subprocess.run` 要捕获命令的标准输出和标准错误（通常显示在控制台上的信息），并赋值给 `result` 函数。

现在，代码将改为如下的格式：

```
import glob
import os
import subprocess

# 获取当前目录下的所有pdbqt文件
pdbqt_files = glob.glob('*.pdbqt')

# 确保结果文件夹存在
results_folder = 'results'
if not os.path.exists(results_folder):
    os.makedirs(results_folder)

# Vina命令的基本部分
vina_command_base = "vina --ligand={} --config=config.txt"
```



```
for file in pdbqt_files:
    # 构建完整的Vina命令
    vina_command = vina_command_base.format(file)
    # 执行vina_command指定的命令，并将该命令的输出以文本形式捕获，然后将这些信息存储在result变量中，以便后续使用或处理。
    results = subprocess.run(vina_command.split(), capture_output=True, text=True)
```

到现在，vina 的输出已经被我们捕获，但我们还需要将其输出为文件。在保存之前，我们需要构建输出文件的路径和文件名。我们定义一个变量“output_file”，使用 os.path.join 函数来构建输出文件的完整路径。

```
output_file = os.path.join(results_folder, file + '_result.txt')
```

这个函数将 result_folder（之前定义的存放结果的文件夹）和文件名（由 file 变量和字符串‘_result.txt’ 拼接而成）组合在一起，从而生成一个路径字符串。

到现在，就可以输出了。

```
with open(output_file, 'w') as file:
    file.write(result.stdout)
```

这里使用了 with 语句和 open 函数来打开文件。open(output_file, 'w') 表示打开 output_file 路径所指向的文件用于写入（‘w’ 模式）。如果文件不存在，它将被创建。使用 with 语句的好处是，它会在代码块执行完毕后自动关闭文件，即使中途发生了异常。as file 表示，这部分将打开的文件对象赋值给变量 file。这样，你就可以通过 file 变量来操作文件。

使用 write 方法将 result.stdout 的内容写入文件，每次循环迭代都会针对不同的 pdbqt 文件生成一个新的输出文件。

完整的代码如下：

```
import glob
import os
import subprocess

# 获取当前目录下的所有pdbqt文件
pdbqt_files = glob.glob('*.pdbqt')

# 确保结果文件夹存在
results_folder = 'results'
if not os.path.exists(results_folder):
    os.makedirs(results_folder)

# Vina命令的基本部分
vina_command_base = "vina --ligand={} --config=config.txt"

for file in pdbqt_files:
    # 构建完整的Vina命令
    vina_command = vina_command_base.format(file)
    # 运行Vina命令并捕获输出
    results = subprocess.run(vina_command.split(), capture_output=True, text=True)
    # 将输出保存到文件
    output_file = os.path.join(results_folder, file + '_result.txt')
    with open(output_file, 'w') as file:
```

```
file.write(results.stdout)
```

3.4.1 vina 断点续算

断点续算 (Checkpointing) 是在长时间运行或资源密集的计算任务中的一个重要概念。它的主要目的是为了效率和可靠性。如果你有一个需要运行数小时甚至数天的任务，中途可能会因为各种原因（如电源中断、系统崩溃、网络问题等）导致中断。断点续算允许程序在中断后从最近的保存点重新开始，而不是从头开始。在处理大量数据或执行大规模计算时，每次从头开始都可能会浪费大量的计算资源和时间。通过保存中间状态，可以在出现问题时从最近的状态恢复，从而优化资源使用。

在大批量的分子对接中，时间往往会花费几天几夜。我们需要为 AutoDock Vina 命令执行添加了断点续算功能。在执行 Vina 命令之前，我们需要脚本检查当前 for 循环中取到的小分子，它的结果输出文件是否已经存在，若已经存在，就说明已经做过了对接，即可以跳过。

最简单的方式是，我们只需要加一个 “if not” 语句。除此之外，我们还加入一些提示语句，使用 print 函数输出。

```
import glob
import os
import subprocess

# 获取当前目录下的所有pdbqt文件
pdbqt_files = glob.glob('*.pdbqt')

# 确保结果文件夹存在
results_folder = 'results'
if not os.path.exists(results_folder):
    os.makedirs(results_folder)

# Vina命令的基本部分
vina_command_base = "vina --ligand={} --config=config.txt"

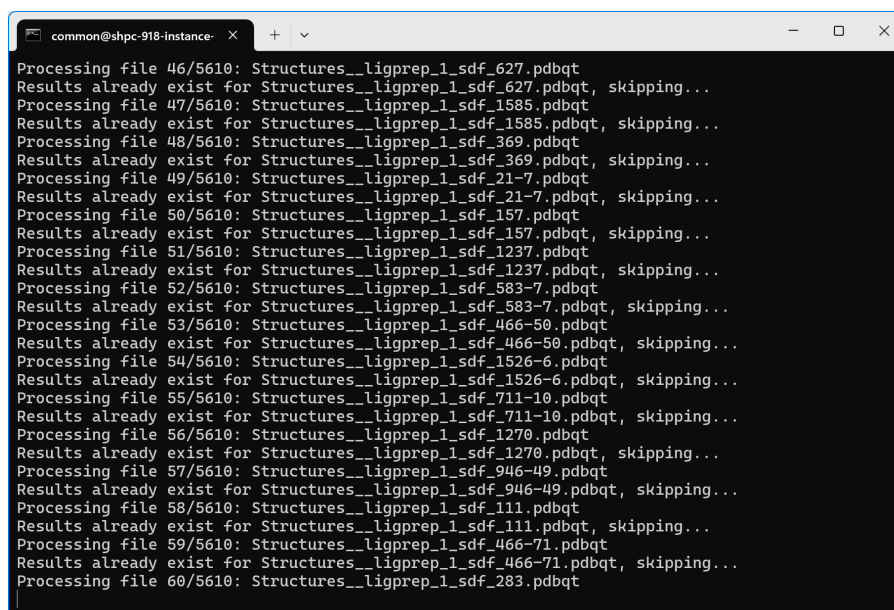
# 追踪当前处理的文件编号
file_number = 0

for file in pdbqt_files:
    file_number += 1 # 增加文件编号
    print(f"Processing file {file_number}/{len(pdbqt_files)}: {file}")

    # 构建结果文件的路径
    output_file = os.path.join(results_folder, file + '_result.txt')

    # 检查结果文件是否已存在
    if not os.path.exists(output_file):
        # 构建完整的Vina命令
        vina_command = vina_command_base.format(file)

        # 运行Vina命令并捕获输出
        results = subprocess.run(vina_command.split(), capture_output=True, text=True)
```



```

common@shpc-918-instance- x + v
Processing file 46/5610: Structures_ligprep_1_sdf_627.pdbqt
Results already exist for Structures_ligprep_1_sdf_627.pdbqt, skipping...
Processing file 47/5610: Structures_ligprep_1_sdf_1585.pdbqt
Results already exist for Structures_ligprep_1_sdf_1585.pdbqt, skipping...
Processing file 48/5610: Structures_ligprep_1_sdf_369.pdbqt
Results already exist for Structures_ligprep_1_sdf_369.pdbqt, skipping...
Processing file 49/5610: Structures_ligprep_1_sdf_21-7.pdbqt
Results already exist for Structures_ligprep_1_sdf_21-7.pdbqt, skipping...
Processing file 50/5610: Structures_ligprep_1_sdf_157.pdbqt
Results already exist for Structures_ligprep_1_sdf_157.pdbqt, skipping...
Processing file 51/5610: Structures_ligprep_1_sdf_1237.pdbqt
Results already exist for Structures_ligprep_1_sdf_1237.pdbqt, skipping...
Processing file 52/5610: Structures_ligprep_1_sdf_583-7.pdbqt
Results already exist for Structures_ligprep_1_sdf_583-7.pdbqt, skipping...
Processing file 53/5610: Structures_ligprep_1_sdf_466-50.pdbqt
Results already exist for Structures_ligprep_1_sdf_466-50.pdbqt, skipping...
Processing file 54/5610: Structures_ligprep_1_sdf_1526-6.pdbqt
Results already exist for Structures_ligprep_1_sdf_1526-6.pdbqt, skipping...
Processing file 55/5610: Structures_ligprep_1_sdf_711-10.pdbqt
Results already exist for Structures_ligprep_1_sdf_711-10.pdbqt, skipping...
Processing file 56/5610: Structures_ligprep_1_sdf_1270.pdbqt
Results already exist for Structures_ligprep_1_sdf_1270.pdbqt, skipping...
Processing file 57/5610: Structures_ligprep_1_sdf_946-49.pdbqt
Results already exist for Structures_ligprep_1_sdf_946-49.pdbqt, skipping...
Processing file 58/5610: Structures_ligprep_1_sdf_111.pdbqt
Results already exist for Structures_ligprep_1_sdf_111.pdbqt, skipping...
Processing file 59/5610: Structures_ligprep_1_sdf_466-71.pdbqt
Results already exist for Structures_ligprep_1_sdf_466-71.pdbqt, skipping...
Processing file 60/5610: Structures_ligprep_1_sdf_283.pdbqt

```

图 3.1: 批量对接成功运行示意图。

```

# 将输出保存到文件
with open(output_file, 'w') as f:
    f.write(results.stdout)
else:
    print(f"Results already exist for {file}, skipping...")

print("Processing complete.")

```

最后的 else 是与之前的 if 并列的（注意看缩进）。这里表示，如果“if”的条件不成立，则运行“else”的代码块，输出“Results already exist for {file}, skipping...”。这里 {file} 的大括号也是字符串格式化的一种，在这里表示用变量 file 来替换。

现在，可以开始批量对接了（保证受体、配体的 pdbqt 文件、config.txt、python 代码都在同一目录下）。在命令行中执行：

```
python mutiple-docking.py
```

Enjoy your docking!

3.5 批量对接全流程

在同一个工作目录下，确保有所有配体的 pdbqt 文件、受体的 pdbqt 文件、config.txt、对接的 python 脚本（代码见下）。在对接之前，可进行单个对接，以确保受体结构无误。

在工作目录新建一个 python 文件（使用 nano 或 RStudio），为 mutiple-docking.py（你可以在 github 仓库中找到。<https://github.com/Marissapy/UESTC-AntiAging-X-BioinformGuide>）内容如下：

```

import glob
import os
import subprocess

# 获取当前目录下的所有pdbqt文件
pdbqt_files = glob.glob('*.pdbqt')

```

```

# 确保结果文件夹存在
results_folder = 'results'
if not os.path.exists(results_folder):
    os.makedirs(results_folder)

# Vina命令的基本部分
vina_command_base = "vina --ligand={} --config=config.txt"

# 追踪当前处理的文件编号
file_number = 0

for file in pdbqt_files:
    file_number += 1 # 增加文件编号
    print(f"Processing file {file_number}/{len(pdbqt_files)}: {file}")
    # 构建结果文件的路径
    output_file = os.path.join(results_folder, file + '_result.txt')
    # 检查结果文件是否已存在
    if not os.path.exists(output_file):
        # 构建完整的Vina命令
        vina_command = vina_command_base.format(file)
        # 运行Vina命令并捕获输出
        results = subprocess.run(vina_command.split(), capture_output=True, text=True)
        # 将输出保存到文件
        with open(output_file, 'w') as f:
            f.write(results.stdout)
    else:
        print(f"Results already exist for {file}, skipping...")

print("Processing complete.")

```

使用 `cd` 切换到工作目录。在命令行中执行：

```
python mutiple-docking.py
```

等待几个小时。当程序运行完毕时，工作目录下新建一个 `python` 文件，执行下面的代码（你可以在 Github 仓库中找到 `affinities.py`。 <https://github.com/Marissapy/UESTC-AntiAging-X-BioinformGuide>）（需要提前 `pip install pandas`）：

```

# -*- coding: utf-8 -*-
import os
import pandas as pd
# 指定目录路径
dir_path = r"/results"
# 遍历目录及子目录下所有.log文件，并提取最小affinity值
affinities = []
for root, dirs, files in os.walk(dir_path):
    for file in files:
        if file.endswith(".txt"):
            with open(os.path.join(root, file), "r") as log_file:
                lines = log_file.readlines()
                affinity = None

```



```

    for line in lines:
        if line.startswith(" 1"):
            affinity = float(line.split()[1])
            break
        if affinity is not None:
            ligand = file.split(".")[0]
            affinities.append((ligand, affinity))
# 将数据存储在Pandas数据框中
df = pd.DataFrame(affinities, columns=["Ligand", "Affinity"])

# 将数据框输出为CSV文件
df.to_csv("results/affinities.csv", index=False)
print("done")

```

本代码用于遍历每个结果文件，抽取出亲和能结果并输出表格。最后，在 `results` 目录下，你会找到 `affinities.csv`，这便是所有的亲和能。

3.6 DEBUG

在对接程序运行后，请不要随意停止！！如果仅出于测试的角度，请在程序停止后，用 RStudio 删除 `results` 文件夹，以免引起 bug。

```

common@shpc-918-instance-mFce3E25:~$ python dock03/md.py
File "dock03/md.py", line 1
import glob
^
IndentationError: unexpected indent

```

line 1 说明在第一行出了问题。检查发现脚本第一行有缩进。python 脚本的第一行一定要顶格。

```

common@shpc-918-instance-mFce3E25:~$ python dock03/md.py
common@shpc-918-instance-mFce3E25:~$

```

python 代码执行后什么都没输出，说明 vina 报错，而不是 python 报错。检查 `config.txt` 有没有写错。

```

common@shpc-918-instance-mFce3E25:~/dock02$ python md-copy.py
bash: python: command not found

```

将 `python` 改为 `python3`，执行 `python3 xxx.py`。

```

common@shpc-918-instance-mFce3E25:~/dock02$ python3 3.py
Traceback (most recent call last):
File "3.py", line 3, in <module>
import pandas as pd
ModuleNotFoundError: No module named 'pandas'

```

包没安装，使用 `pip install xxxxx`。

第四章 结语

分子对接是一座大山，但只是漫长生物信息学的开头一步，恭喜你已经迈过了坎！
请保持学习！

Yan Pan,

January 30, 2024