GOpt (Global Optimizer)

用户手册

2022 年 10 月 江苏·无锡

目　录

第一章 简介

GOpt（Global Optimizer）群体智能优化平台是由雪浪云与国内计算智能领域知名团队安徽大学张兴义教授课题组共同打造的明珠产品，其由领域主流平台PlatEMO原班人马参与打造，继承了其拓展性强、易于使用等优点，同时GOpt在产业界应用方面有着强大的算力支持以及提供云服务等更大优势。GOpt以雪浪OS为基座，集成领域内百余种最先进的元启发式算法，构建统一的算法评价体系；实现一站式的优化模板复用与快速扩展及适配；采用完全自主可控的计算架构与超强算力，助力关键技术国产化替代。

GOpt平台建立产学研“三界一体”的全方位产品、生态、应用布局。在智能制造领域，提供了适用范围广、优化性能强的元启发式算法，解决关键场景中的核心业务决策的问题；在人才培养领域，提供了可视化交互界面与强大的算法学习、测试功能，提升了代码的复用性，降低了算法的测试代价；在科学研究领域，提供了学术界最前沿的算法库及完善的拓展功能，实现研究人员对于科研工具与实验平台的多元需求。

1.1GOpt 产品功能简介

本平台是一个用于求解复杂优化问题的工具，它的输入是一个优化问题，输出是在该优化问题上得到的最优解。一个优化问题满足以下定义：

$$\begin{aligned} \min_{\mathbf{x}} \ & \mathbf{f}(\mathbf{x}) = \left(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_M(\mathbf{x})\right) \\ \text{s.t.} \ & g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots, g_K(\mathbf{x}) \leq 0 \end{aligned}$$

其中 $\mathrm{x} = (\mathrm{x}_1, \mathrm{x}_2, \ldots \mathrm{x}_\mathrm{D}) \in \Omega$ 表示该问题的一个解或决策向量，它由D个决策变量组成，其中每个决策变量可能被限制为实数、整数或二进制数等。Ω 表示该问题的搜索空间，它由下界 $\mathrm{l}_1, \mathrm{l}_2, \ldots \mathrm{l}_\mathrm{D}$ 和上界 $\mathrm{u}_1, \mathrm{u}_2, \ldots \mathrm{u}_\mathrm{D}$ 构成，即任意决策变量始终满足 $\mathrm{l}_\mathrm{i} \leq \mathrm{x}_\mathrm{i} \leq \mathrm{u}_\mathrm{i}$。$\mathrm{f}_1(\mathrm{x}), \mathrm{f}_2(\mathrm{x}), \ldots, \mathrm{f}_\mathrm{M}(\mathrm{x})$ 表示该解的M个目标函数值，$\mathrm{g}_1(\mathrm{x}), \mathrm{g}_2(\mathrm{x}), \ldots, \mathrm{g}_\mathrm{K}(\mathrm{x})$ 表示该解的K个约束违反值。

为了定义一个优化问题，用户至少需要输入以下内容：

- 决策变量的数目 D 和目标函数的数目 M；

- 决策变量的编码方式（实数、二进制数或序列）；
- 决策变量的下界 $l_1, l_2, \ldots l_D$ 和上界 $u_1, u_2, \ldots u_D$；
- 至少一个目标函数 $f_1(\mathbf{x})$。

为了更精准地定义问题，用户还能输入以下内容：

- 多个目标函数 $f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_M(\mathbf{x})$；
- 多个约束函数 $g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots, g_K(\mathbf{x})$；
- 解的初始化函数；
- 无效解的修复函数；
- 各函数计算中使用到的数据集（一个全局常量）。

以上函数均指的是代码函数而非数学函数，即它需要有符合规定的输入和输出，但不需要有显式的数学表达式。此外，用户还能定义与优化算法相关的内容，通过选择合适的算法和参数设置以提升优化效果。

1.2 平台登陆

平台登陆链接：http://gopt.xuelangyun.com/gopt/login

1.3 联系我们

用户在使用过程中，遇见产品登陆、使用等相关问题，可添加下方微信，咨询。

联系我们

第二章 使用方法

用户可通过 http://gopt.xuelangyun.com/gopt/login 进行登录，如未注册账号可在页面提示下进行注册，登陆后可以进入如下界面：

图一 GOpt 首页

初次使用，可以通过<新建项目>新建问题求解。已有项目，可通过双击项目名称进入项目。

进入项目后首先进入的是问题的选择界面，在该界面可以选择内置的测试问题进行测试，通过图二中 A 区域对测试问题进行筛选。同时也可以自定义创建问题。

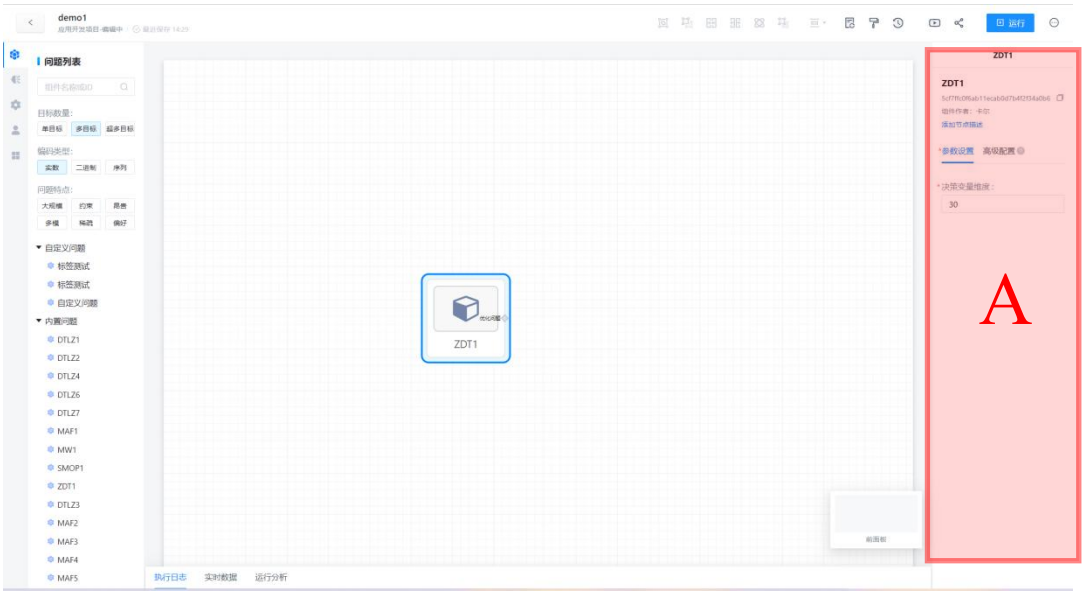
图二 GOpt 项目页面

2.1 求解内置问题

内置问题是指由开发者设计好的、用于测试算法性能优劣的问题。不同的问题拥有很强的针对性。这些问题涵盖范围广，自变量的维度从几到上万不等，目标维度从单目标、多目标乃至超多目标。这些内置问题能很好的用于测试算法在不同情况下的性能优劣，以及辅助初学者掌握软件。GOpt 平台目前涵盖了近 100+ 不同的内置问题，覆盖面积广，满足用户各个使用需求。

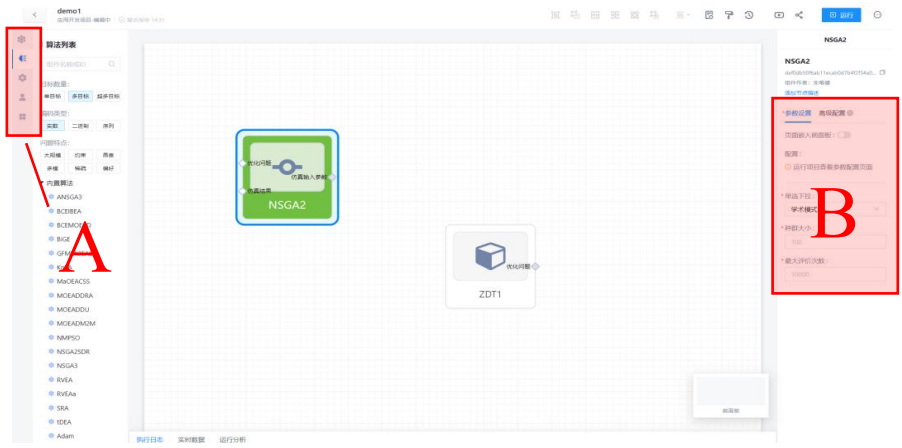
下面是如何求解内置问题。

1. 从左侧内置问题的列表中选定一个<内置问题>，通过拖拽的方式将其拖入后面板（图二 B 区域）中，然后选中该测试问题（图三以 ZDT1 为例），右侧（图三 A 区域）会出现该测试问题的一些可设置参数，如决策变量维度等，部分测试问题可以设置其目标数。



图三 问题筛选页面

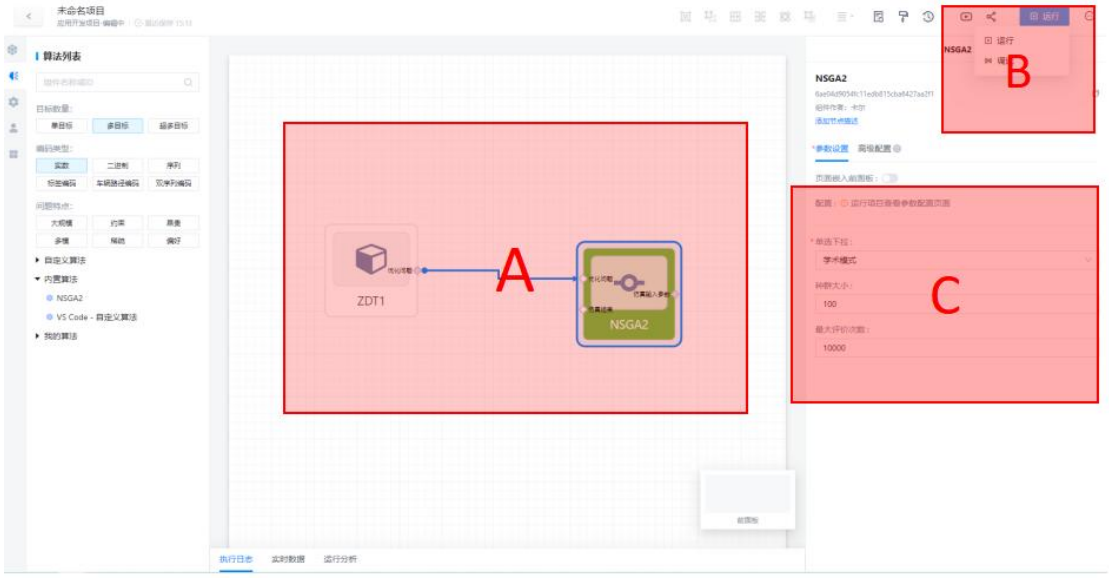
2. 选定问题后，从最左侧工具栏（图四 A 区域）按钮进入算法筛选界面，同样是通过左侧的标签按钮来筛选对应属性的算法。在筛选算法时请务必保证第一步中选择的问题与选择的算法标签一致。选定某个算法后，可以通过拖拽的方式将选定的算法拖入后面板中，然后点击选中算法组件，并配置算法相应的参数（图四 B 区域）。每个算法都有自己的默认参数，可以无需设置。



图四 左侧工具栏

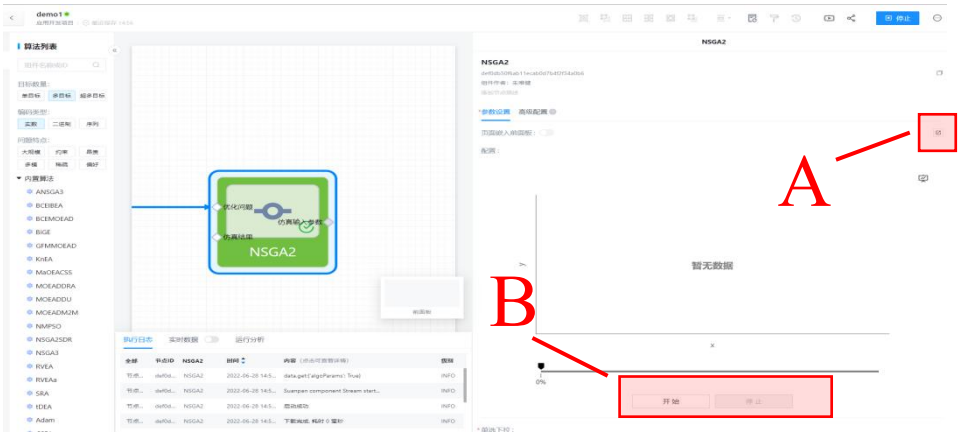
3. 将测试问题与算法的参数配置完成后，如图五，在区域 C 中配置相关参数后，将算法组件左上角的输入接口与测试问题组件右边的输出接口相连（如图五 A 区域所示），并点击页面中右上角的运行按钮（图五 B 区域），即可运行程序。

注意：当测试问题与算法的标签不匹配时连线上将会出现提示。



图五 组件相连

4. 项目运行成功后再次选中算法组件，将会看到如下图所示界面，右边算法组件的拓展部分作为运行效果的展示，可以通过点击下图中红色框内的<拓展>按钮（图六 A 区域）将该效果展示在另外一个页面中打开，以达到更好的观看效果。



图六 算法运行后展示

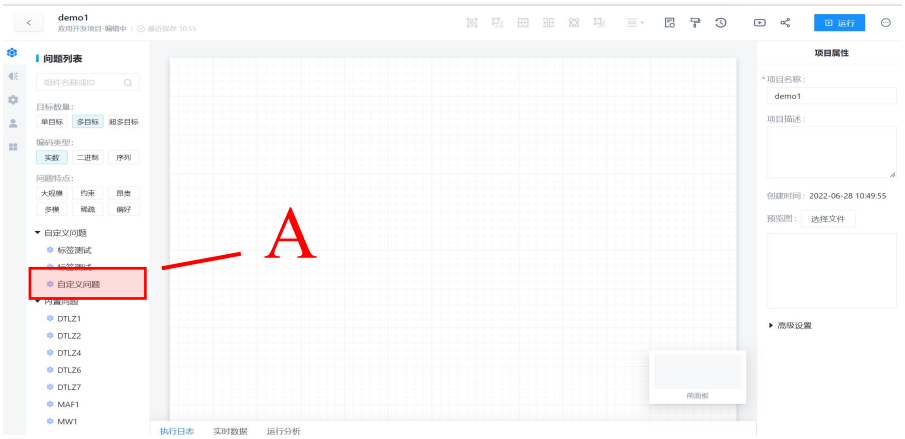
5. 点击<开始>按钮（图六 B 区域）即可在 ZDT1 上运行 NSGA2 算法

2.2 求解自定义问题

在实际生产中，往往会遇到各种各样的实际问题，如何使用 GOpt 平台去求解实际问题是平台的研究重点。GOpt 平台设计了一个自定义问题组件，在该组件中用户可以自行设计符合实际用户需求的实际问题，再从 GOpt 平台中选择合适的算法去求解，最终得到所需的解决方案。

下面是如何求解自定义问题。

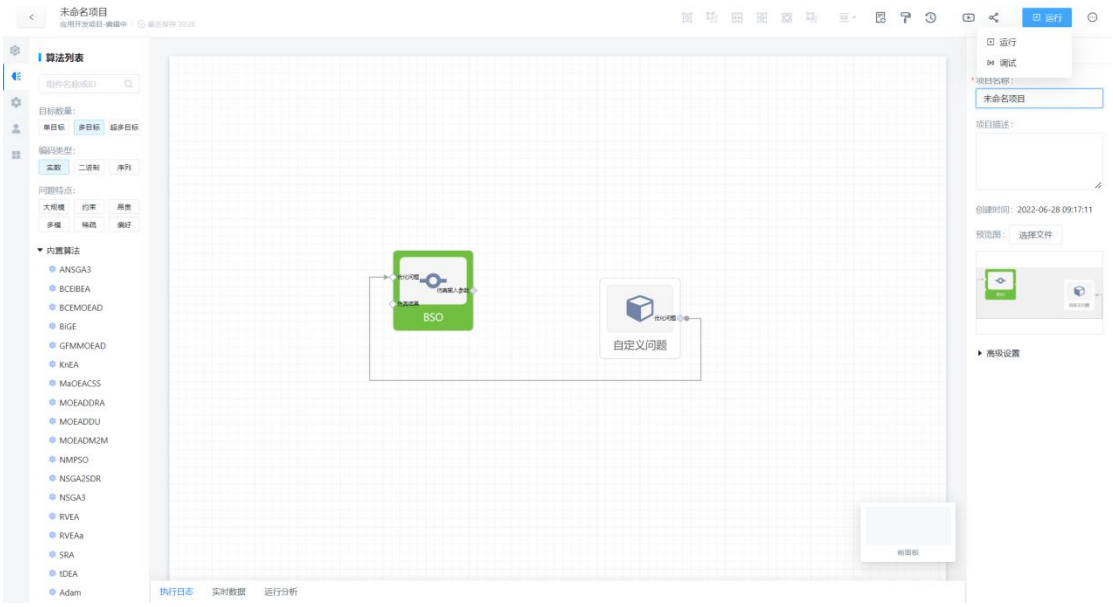
1. 在问题列表选择<自定义问题组件>（图七 A 区域）并将其拖入后面板中；



图七 问题列表界面

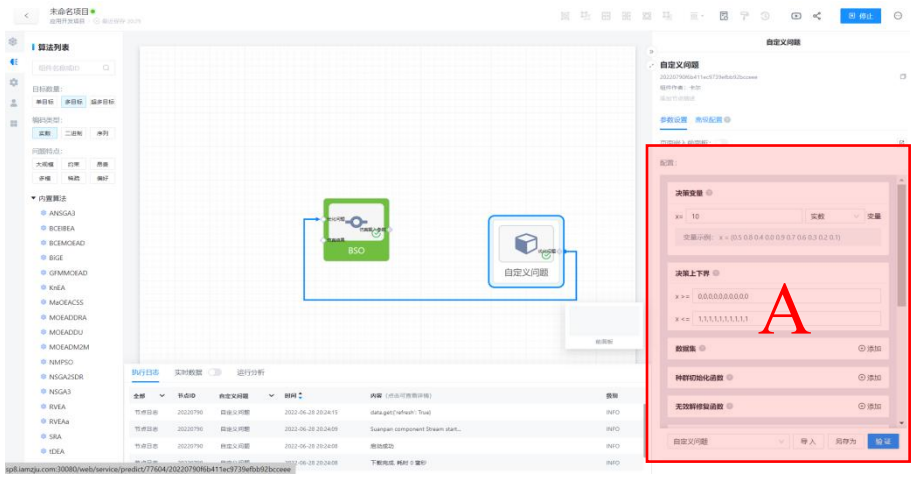
2. 在算法筛选界面中选择需要定义问题类型匹配的算法拖入到后面板中，并将

算法组件与自定义问题组件的输入输出相连。



图八 算法列表界面

3. 运行该项目，在项目运行成功后，选中自定义问题组件，自定义问题组件的界面将会在界面右侧（图九 A 区域）显示，建议点击<拓展>按钮到另一个界面进行问题的定义。

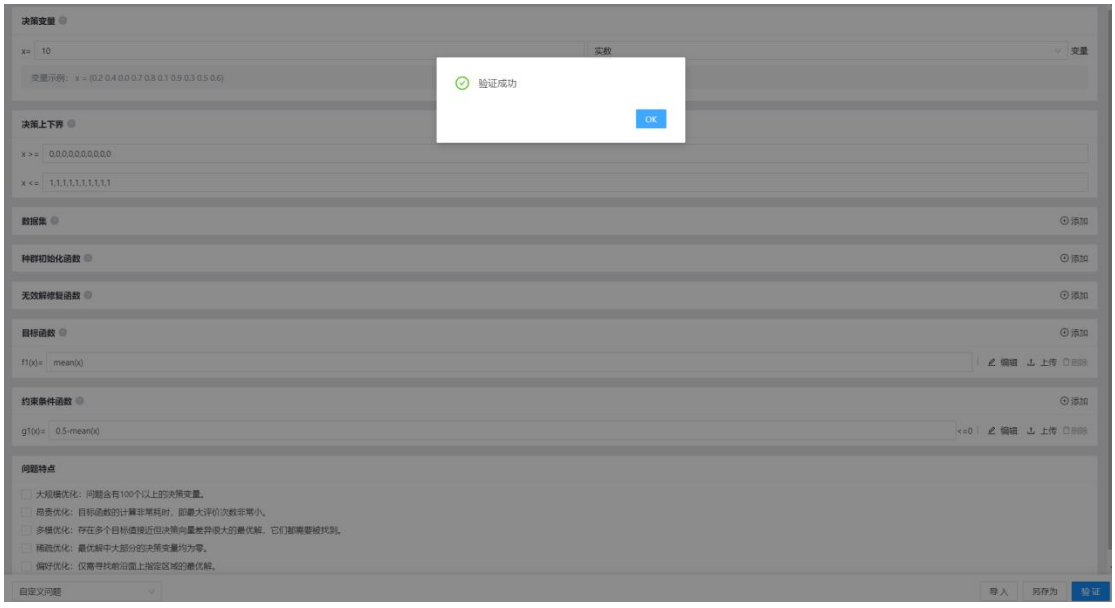


图九 自定义问题组件详情

4. 对于一个自定义问题，至少需要指定决策变量长度、决策变量上下界、目标值函数才能构建一个最基本的用户自定义问题。在该组件的左下角，给出了把八种基本自定义问题的模板，以供用户参考。

图十 自定义问题组件提供模板类型

5. 对于自定义问题，用户可以根据需要通过<新增>或者<删除>按钮来调整问题的目标数、约束数等内容，值得注意的是，在定义目标时，定义函数的内容需要满足 python 科学运算的语法，并且该目标函数是基于一个行向量（即一个长度为 D 的矩阵）进行运算。在完成自定义问题后，用户可以通过点击自定义问题组件详情界面右下角的<验证>按钮来进行问题的验证，若问题定义正确则会显示验证成功，此时可以转到项目界面，点击算法组件来运行算法。



图十一 验证问题成功页

6. 此外在各函数部分，可以上传.py 文件用作函数求解。

Examples

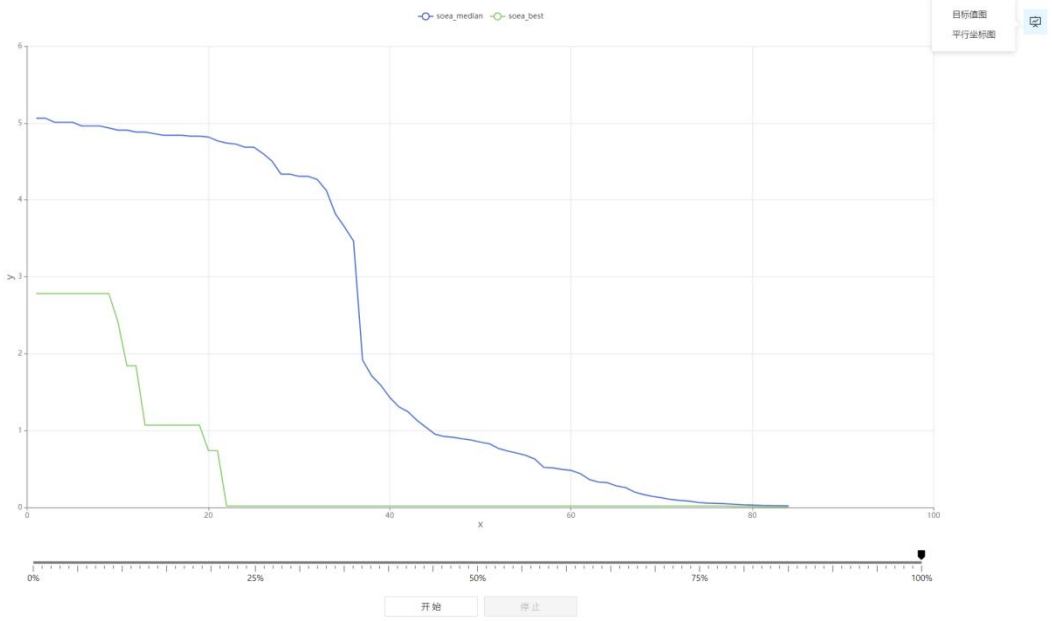
>>> def main(x, data):

>>> return x[0] + data[0]

这里计算函数值时，每次计算一个解的函数值，其中 x 是种群中的一个解，是一个一维长度为 D 的 ndarray 数组（D 是决策变量长度）；data 是传入的数据集列表，类型为 list，data[0]表示第一个数据集、data[1]是第二个数据集...，且如果上传的数据集是二维的，那么 data[0]是一个二维的 ndarray。

该文件中可包含多个其他函数用于辅助计算，最终 GOpt 会以 main（）的返回值为准。此外，py 文件中导包，默认导入 numpy 包，若使用到其他第三方库，可在自定义组件的部分，将所需的库及版本添加即可。若不指定版本，使用 NULL，如：{"pydoe":null}。

转到算法组件的详情页后，点击开始后算法即可求解自定义问题，结果展示如下图所示。



图十二 算法运行展示

第三章 详细说明

3.1 标签

GOpt 的内置算法以及问题都带有标签，所有的标签列举如下：

表 1 标签说明

标签	描述
单目标	问题含有一个目标函数
多目标	问题含有两或三个目标函数
超多目标	问题含有三个以上目标函数
实数	决策变量为实数或整数
二进制	决策变量为二进制数
序列	决策变量构成一个序列
大规模	问题含有 100 个以上的决策变量
约束	问题含有至少一个约束
昂贵	目标函数的计算非常耗时，即最大评价次数非常小
多模	存在多个目标值接近但决策向量差异很大的最优解，它们都需要被找到
稀疏	最优解中大部分的决策变量均为零
偏好	仅需寻找前沿面上指定区域的最优解
无	上述各标签后加无，表示可求解无该标签的问题

每个算法可能含有多个标签集合，这些集合的笛卡尔积构成该算法可求解的所有的问题类型。例如当标签集合为<单目标><实数><约束/无>时，表示该算法可求解带或不带约束的单目标连续优化问题；若标签集合为<单目标><实数>，表示该算法只能求解无约束问题；若标签集合为<单目标><实数><约束>，表示该算法只能求解有约束问题；若标签集合为<单目标><实数/二进制>，表示该算法可以求解连续或二进制优化问题。每个算法和测试问题都需要被添加至少一个标签，否则它将不会在图形界面的列表中出现。

3.2 内置问题组件

表 2 问题的可选参数

参数	赋值方式	描述
目标数	用户和默认值	问题的目标数
决策变量长度	用户和默认值	问题的变量数
其他参数	用户和默认值	问题的特定参数

‘目标数’：设置问题的目标函数个数。不同问题函数表达式不同，设置目标数，可以产生对应数量的目标函数，至少需要定义一个目标函数。

‘决策变量长度’：定义待优化的决策向量 $\mathbf{x} = (x_1, x_2, \ldots x_D) \in \Omega$ 的格式，其中决策变量的长度 D 需要用户设置或者使用内置问题的默认值。

‘其他参数’：内置问题中包含某些内置问题有特定参数，用户可以自行根据提示设置其内容，也可使用默认值。

3.3 算法组件

表 3 算法的可选参数

参数	赋值方式	描述
模式	用户	学术模式与工程模式
种群大小	用户和默认值	求解该问题的算法的种群大小
最大评价次数	用户和默认值	求解该问题可使用的

		最大评价次数
其他参数	用户和默认值	算法的特定参数

‘模式’：用户通过下拉框设置算法的模式。其中学术模式，用户可以使用各种内置问题组、使用自定义问题组件自行设置问题或者使用 web input 组件通过输入自定义问题的各个参数进而求解自定义问题。

‘种群大小’：求解该问题的算法的种群大小，即初始解的数量。

‘最大评价次数’：求解该问题可使用的最大评价次数，算法中每求解一次目标值，评价次数加 1，用户可设置最大评价次数，当算法达到最大评价次数，算法终止。

‘其他参数’：内置算法中某些算法有特定的参数，这些参数对算法的结果有一定的影响，用户可以根据提示自行设置其内容，也可使用默认值。

第四章 自定义问题组件

4.1 自定义问题组件创建

对于一个自定义问题，自定义组件提供了八种类型的自定义问题模板以供用户使用。 用户根据需求可以选择不同类型去定义问题，也可以导入问题。

A）若用户使用组件内各模块进行问题定义，具体过程可详见 2.2 节。

图十三 自定义问题组件

用户首先在区域 A 中设置决策变量长度和编码类型；其次在区域 B 中给定决策变量的上下界；对于一个复杂问题，用户可在区域 C 中编辑或上传数据集，格式为 CSV/TXT 固定格式读取矩阵数据；用户在区域 D、E、F、G 中确定各类函数。

注：各类函数的编写需要遵循 Python 语言。

在定义好问题后，可以点击<验证>按钮，自定义问题组件会自动验证该问题是否合法。另外用户也可将定义好的问题按 json 类型导出。用户可以通过点击<另存为>按钮，GOpt 平台将自动将定义好的问题按照 json 类型导出。

下面是对各模块含义的详细介绍：

‘决策变量’：定义待优化的决策向量 $\mathbf{x} = (x_1, x_2, \ldots x_D) \in \Omega$ 的格式，包括变量的数目 D 以及变量的编码方式。

‘决策上下界’：决策变量的取值范围，即搜索空间，定义所有决策向量 $\mathbf{x} = (x_1, x_2, \ldots x_D) \in \Omega$ 的上下界范围，其中上、下界均为 1×D 的向量。当一个决策变量越界时，它将被自动设为相应的边界值。当决策变量为二进制或序列编码时，该设置无效。

‘数据集’：定义所有函数计算中使用到的复杂数据，它可以是一个任意类型的变量、或是一个仅包含矩阵数据的文本文件。数据集将作为所有函数的第二个输入参数传入，并在函数中任意使用。数据集可以被省略，此时所有函数仅有一个输入参数。

‘种群初始化函数’：定义一个种群初始化函数，用于产生多个不同的初始解。该函数的输入为产生的决策向量的数目 N(一个正整数)，输出为产生的初始决策向量的集合（一个 N*D 矩阵)。通常，优化算法在开始时会调用该函数。该函数可以被省略，此时优化算法将根据变量的类型随机地产生初始决策向量。

‘无效解修复函数’：定义一个修复函数，用于修复非法的解。该函数的输入为一个决策向量 x (一个 1×D 向量)，输出为修复后的决策向量（一个 1×D 向量)。通常，优化算法在产生解之后会调用该函数。该函数可以被省略，此时将不会进行修复操作。还是进行默认的修复操作？

‘目标函数’：定义一或多个目标函数，用于计算解的目标值。每个目标函数的输入为一个决策向量 x (一个 1×D 向量)，输出为解在该函数上的目标值（一个标量)。通常，优化算法在产生解之后会调用该函数。至少需要定义一个目标函数。所有目标函数均为最小化目标。

‘约束函数’：定义零或多个约束函数，用于计算解的约束值。每个约束函数的输入为一个决策向量x(一个 1×D 向量)，输出为解在该函数上的约束违反值（一个标量)。通常，优化算法在产生解之后会调用该函数。该函数可以被省略。当且仅当约束违反值小于等于零时，该约束条件被视为满足。

B）若用户导入问题文件，文件需为 json 类型，点击自定义问题组件下方的导入按钮，选择问题文件即可。提供一个实例：


```
{
    "name": "custom_problem",
    "encoding": "real",
    "n_var": 30,
    "lower": "0",
    "upper": "1",
    "variables": {
        "g": "1 + 9 * mean(x[1:])",
        "h": "1 - sqrt(x[0] / g)"
    },
    "initFcn": [],
    "decFcn": [],
    "objFcn": [
        "x[0]",
        "g * h"],
    "conFcn": []
```

}

Json 文件中各变量介绍如下表所示：

表 4 自定义问题的可选参数

参数名	数据类型	默认值	描述
'name'	字符串	'custom_problem'	问题名称
'encoding'	字符串	'real'	编码方式
'n_var'	整型数字	30	决策变量维度
'objFcn'	字符串列表	[“mean(x)”]	问题的目标函数,各个目标中间以逗号分隔
'conFcn'	字符串列表	[]	问题的约束,各个约束中间以逗号分隔;
'lower'	数值或行向量	-np.inf	变量的下界
'upper'	数值或行向量	np.inf	变量的上界
'initFcn'	列表	[]	种群初始化函数
'decFcn'	列表	[]	无效解修复函数
'variables'	列表	[]	中间变量

’encoding’表示问题的编码方式，它的值可以是’real’（实数或整数变量）、’binary’（二进制变量）或’permutation’（序列变量）。算法针对不同的编码方式会使用不同的算子来产生子代。

'objFcn'表示问题的目标函数列表，列表中每一个字符串代表着一个目标函数的计算式，其中由 x 代表决策变量，该计算式需要满足 python 数学运算的语法，该列表的长度至少需要为 1。

'conFcn'表示问题的约束列表，列表中每一个字符串代表着一个约束函数的计算式，该计算式需要满足 python 数学运算的语法，当且仅当约束违反值小于等于零时，约束被满足，如有多个约束，以逗号分隔。

'lower'表示决策变量的下界，它仅在'encoding'的值为'real'时生效，可以缺省，默认为负无穷大。

'upper'表示决策变量的上界，它仅在'encoding'的值为'real'时生效，可以缺省，默认为无穷大。

'initFcn'表示种群初始化函数，它代表着种群的初始化方式，可以缺省，默认在决策变量的范围内随机初始化。

'decFcn'表示无效解修复函数，它代表遇到无效解（决策变量越界）时对解进行修复的方式，该函数会在计算目标函数前被调用，可以缺省，默认对越界值赋以较近的上界或下界。

'variables'表示中间变量，它是一个列表，其中的值应该以 key-value 的形式出现，当目标值函数过于复杂或者在其他计算式中有着公共的部分，可以写做中间变量，value 对应中间变量的字符串，key 可以在其余计算式中进行引用。

4.2 自定义问题组件求解实际问题实例

该章节具体介绍如何使用自定义问题组件设计自定义问题，并选择合适的算法求解。主要分为 2 个步骤。

第一步：问题分析：

以一个车辆性能优化问题DDMOP1问题为例，该问题是一个轿车驾驶室设计，具有 11 个决策变量和 9 个目标，具体细节参考论文 https://sci-hub.wf/https://ieeexplore.ieee.org/document/5196713。它是

一个超多目标的实际问题，多个目标之间的优化难以平衡，如何使这些目标尽可能的达到最优是难点所在。决策变量包括车身的尺寸和固有频率的界限，例如B柱内部的厚度、车身侧面内部的厚度、门梁的厚度和护栏高度。 同时，这九个指标表征了汽车驾驶室的性能，如汽车的重量、燃油经济性、加速时间、不同速度下的道路噪声和汽车的宽敞程度。所谓的决策变量在该问题中就是车身的尺寸和固有频率的界限等，决策变量长度是这些信息的数量，上下界是这些信息的范围，如车身尺寸等等；该问题的目标是针对汽车驾驶室的性能，如汽车的重量、燃油经济性、加速时间、不同速度下的道路噪声和汽车的宽敞程度等，其中每个目标函数对应一个目标。

其中 11 个决策变量包括 7 个决策变量（x_1 to x_7）和 4 个随机参数（x_8 to x_{11}）：

x_1：B 柱内部厚度$(0.5 \le x_1 \le 1.5)$；

x_2：B 柱加强件的厚度$(0.45 \le x_2 \le 1.35)$；

x_3：地板内侧厚度$(0.5 \le x_3 \le 1.5)$；

x_4：横梁厚度$(0.5 \le x_4 \le 1.5)$；

x_5：门梁厚度$(0.875 \le x_5 \le 2)$；

x_6：车门腰线加强件的厚度$(0.4 \le x_6 \le 1.2)$；

x_7：车顶纵梁的厚度$(0.3 \le x_7 \le 1.2)$；

x_8：B 柱内部材料$(-1.655 \le x_8 \le 2.345)$；

x_9：地板内侧材料$(-1.808 \le x_9 \le 2.192)$；

x_{10}：障碍物高度$(-2 \le x_{10} \le 2)$；

x_{11}：障碍位置$(-2 \le x_{11} \le 2)$。

括号中的数量表示每个变量的范围。待优化目标的数学表达式表述如下：

$$f_1(\mathbf{x}) = 1.98 + 4.9x_1 + 6.67x_2 + 6.98x_3 + 4.01x_4 + 1.78x_5 + 0.00001x_6 + 2.73x_7$$

$$f_2(\mathbf{x}) = 1.16 - 0.3717x_2x_4 - 0.00931x_2x_{10} - 0.484x_3x_9 + 0.01343x_6x_{10}$$

$$f_3(\mathbf{x}) = 0.261 - 0.0159x_1x_2 - 0.188x_1x_8 - 0.019x_2x_7 + 0.0144x_3x_5 + 0.87570.001x_5x_{10} + 0.08045x_6x_9 + 0.00139x_8x_{11} + 0.00001575x_{10}x_{11}$$

$$f_4(\mathbf{x}) = 0.214 + 0.00817x_5 - 0.131x_1x_8 - 0.0704x_1x_9 + 0.03099x_2x_6 - 0.018x_2x_7 + 0.0208x_3x_8 + 0.121x_3x_9 - 0.00364x_5x_6 + 0.0007715x_5x_{10} - 0.0005354x_6x_{10} + 0.00121x_8x_{11} + 0.00184x_9x_{10} - 0.018x_2^2$$

$$f_5(\mathbf{x}) = 0.74 - 0.61x_2 - 0.163x_3x_8 + 0.001232x_3x_{10} - 0.166x_7x_9 + 0.227x_2^2$$

$$f_6(\mathbf{x}) = 109.2 - 9.9x_2 + 6.678x_3 + 0.1792x_{10} - 9.256x_1x_2 - 12.9x_1x_8 - 11x_2x_8 + 0.1107x_3x_{10} + 0.0207x_5x_{10} + 6.63x_6x_9 - 17.75x_7x_8 + 22x_8x_9 + 0.32x_9x_{10}$$

$$f_7(\mathbf{x}) = 4.72 - 0.5x_4 - 0.19x_2x_3 - 0.0122x_4x_{10} + 0.009325x_6x_{10} + 0.000191x_{11}^2$$

$$f_8(\mathbf{x}) = 10.58 - 0.674x_1x_2 - 1.95x2x8 + 0.02054x_3x_{10} - 0.0198x_4x_{10} + 0.028x_6x_{10}$$

$$f_9(\mathbf{x}) = 16.45 - 0.489x_3x_7 - 0.843x_5x_6 + 0.0432x_9x_{10}$$

对于此类复杂实际问题，用户明确优化目标和各个决策变量后，需要根据优化目标确定目标函数。

第二步：创建自定义问题组件

根据第一步中问题分析得到结果，创建自定义问题组件，将该问题在自定义问题组件上实现。

下面是如何使用自定义问题组件实现车辆性能优化问题。

初始具体操作流程可见章节 2.2 图九部分。因为车辆性能优化问题 DDMOP1 是一个超多目标问题，用户可在自定义问题组件下方的 8 个基本类型中选择“超多目标问题”。对于DDMOP1，事先知道其11个决策变量的长度、范围，即车身的尺寸和固有频率的界限等，用户可在章节 3.4 图十三中区域 A 和 B 中决策变量和决策上下界一栏中将之输入。如图十四所示：

图十四 决策变量长度及范围输入

该实际问题未用到其他数据，数据集一栏可省略。

对于该问题种群初始化函数及无效解修复函数可使用算法内置函数。这里说的种群初始化函数指的是随机生成一组数据的方法；无效解修复函数是指对于上述所限定的决策变量范围，若产生的数据不符合该范围时，使用哪些函数去修复数据。

目标函数这一栏用户需要将上述车辆性能优化问题DDMOP1的 9 个函数表达式通过 python 语句编译并输入至章节 3.4 图十三区域 F 中。

每个目标函数输入内容如下：

f1(x) = 1.98 + 4.9 * x[0] + 6.67 * x[1]+ 6.98 * x[2]+ 4.01 * x[3]+ 1.78 * x[4]+ 0.00001 * x[4]+ 2.73 * x[1];

f2(x) = 1.16 - 0.3717 * x[1] * x[3] - 0.00931 * x[1] * x[9] - 0.484 *x[2] * x[8] + 0.01343 * x[5] * x[9]

f3(x) = 0.261 - 0.0159 * x[0] * x[1] - 0.188 * x[0] * x[7] - 0.019 * x[1] * x[6] + 0.0144 * x[2] * x[4] + 0.87570 * x[4] * x[9] + 0.08045 * x[5] * x[8] + 0.00139 * x[7] * x[10] + 0.00001575 * x[9] * x[10]

f4(x) = 0.214 + 0.00817 * x[4] - 0.131 * x[0] * x[7] - 0.0704 * x[0] * x[8] + 0.03099 * x[1] * x[5] - 0.018 * x[1] * x[6] + 0.0208 * x[2] * x[7] + 0.121 * x[2] * x[8] - 0.00364 * x[4] * x[5] + 0.0007715 * x[4] * x[9] - 0.0005354 * x[5] * x[9] + 0.00121 * x[7] * x[10] + 0.00184 * x[8] * x[9] - 0.018 * x[1] ** 2

f5(x) = 0.74 - 0.61 * x[1] - 0.163 * x[2] * x[7] + 0.001232 * x[2] * x[9] - 0.166 * x[6] * x[8] + 0.227 * x[1] ** 2

f6(x) = 109.2 - 9.9 * x[1] + 6.768 * x[2] + 0.1792 * x[9] - 9.256 * x[0] * x[1] - 12.9 * x[0] * x[7] - 11 * x[1] * x[7] + 0.1107 * x[2] * x[9] + 0.0207 * x[4] * x[9] + 6.63 * x[5] * x[8] - 17.75 * x[6] *x[7] + 22 * x[7] * x[8] + 0.32 * x[8] * x[9]

f7(x) = 4.72 - 0.5 * x[3] - 0.19 * x[1] * x[2] - 0.0122 * x[3] * x[9] + 0.009325 * x[5] * x[9] + 0.000191 * x[10] ** 2

f8(x) = 10.58 - 0.674 * x[0] * x[1] - 1.95 * x[1] * x[7] + 0.02054 * x[2] * x[9] - 0.0198 * x[3] * x[9] + 0.028 * x[5] * x[9]

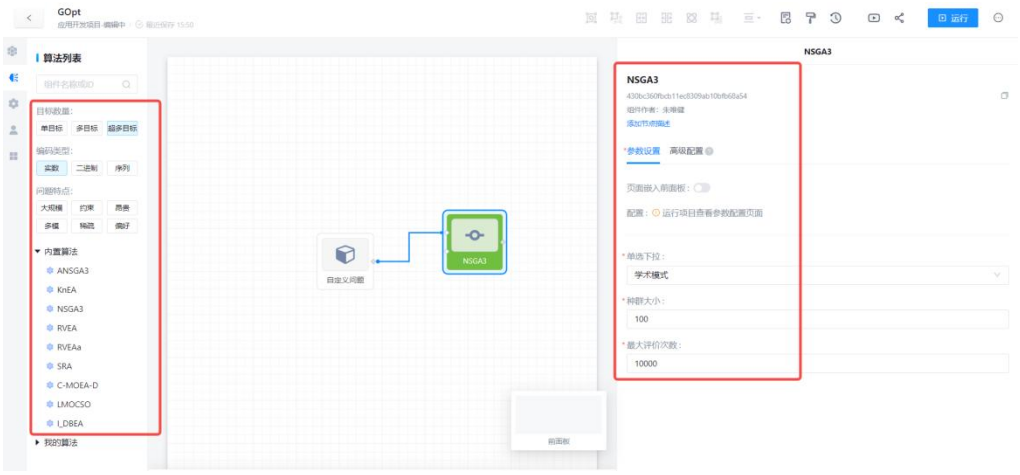
f9(x) = 16.45 - 0.489 * x[2] * x[6] - 0.843 * x[4] * x[5] + 0.0432 * x[8] * x[9]

将其输入至章节 3.4 图十三区域 F 中后如图十五：

图十五 目标函数

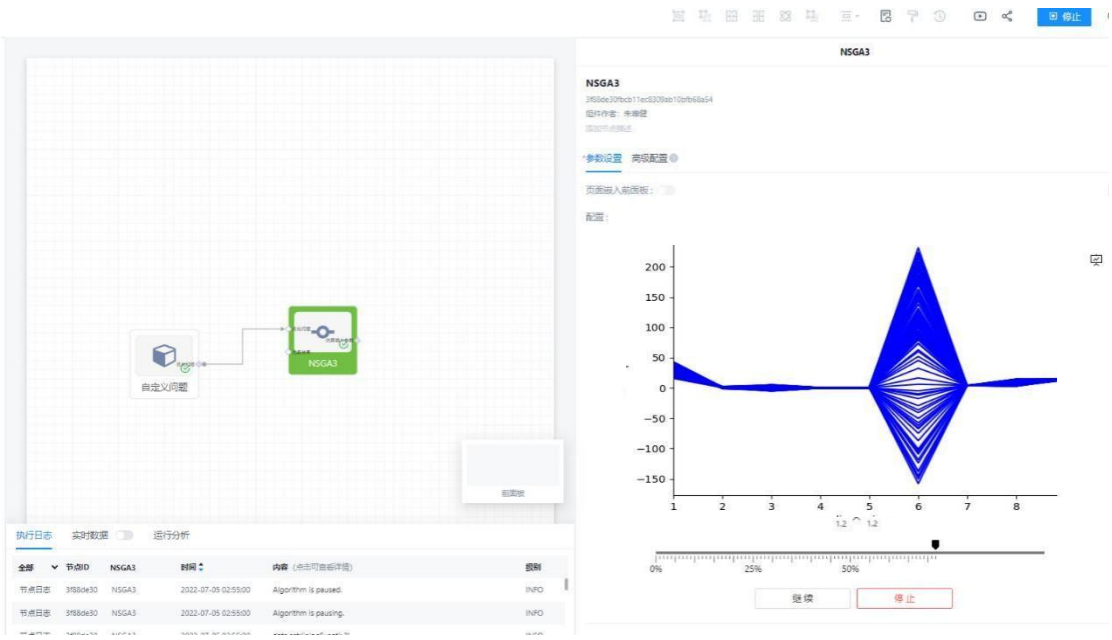
其中 9 个目标函数分别对应该问题所需要优化的 9 个目标。该实际问题约束在目标函数构建时已将其转化成目标，故约束条件函数这一栏可省略。

最后用户需要对该类问题赋予问题特点，以方便 GOpt 推荐合适算法。如车辆性能优化问题 DDMOP1 属于超多目标实数编码问题，GOpt 可以推荐一些合适算法，具体操作为章节 2.2 图八所示。勾选了超多目标和实数编码后，GOpt 推荐的算法如图十六，用户可选取其中任意一算法去求解该问题，也可以尝试多个算法以发现更好的优化结果。



图十六 GOpt 算法推荐及算法组件参数输入

如选取算法NSGA3，具体操作为章节 2.2 所示，将该算法组件拉至前面板与自定义问题组件相连，根据 NSGA23 算法组件参数提示，输入 NSGA3 的一些算法参数如图十六，点击运行，再在 GOpt 平台编译结束后，点击 NSGA3 算法组件中的开始按钮，此时用户在自定义问题组件中定义的实际问题的各个信息以 Json 类型传至算法组件，算法组件接收到自定义问题组件的信息后，算法开始运行，并将优化过程中数据信息实时显示，让用户更 直观的看见优化过程，如图十七：



图十七 算法优化过程

对于该车辆性能优化问题，章节 3.5 开头部分论文中对于该问题提供了真实最优解，如图十八：

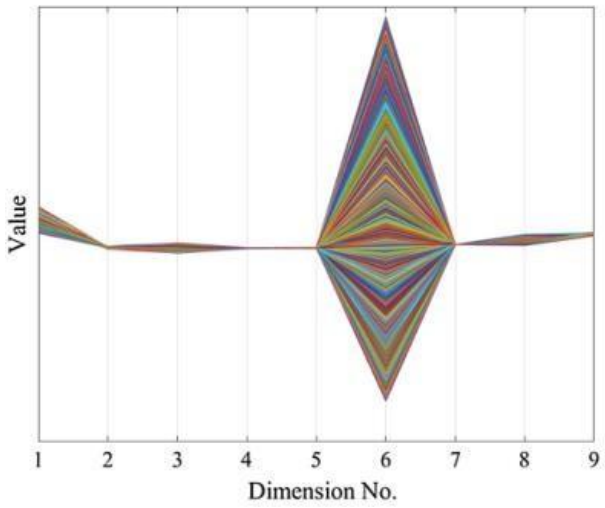


图 十八 DDMOP1 真实最优解

对比GOpt推荐算法NSGA3结果，可以看出GOpt可以很好的得到一组接近真实最优解的解。

第五章 自定义算法组件

5.1 进入算法定义环境

下面是如何创建自定义算法组件的步骤。

1）登录到 GOpt 打开 GOpt 平台网站，登录账号。

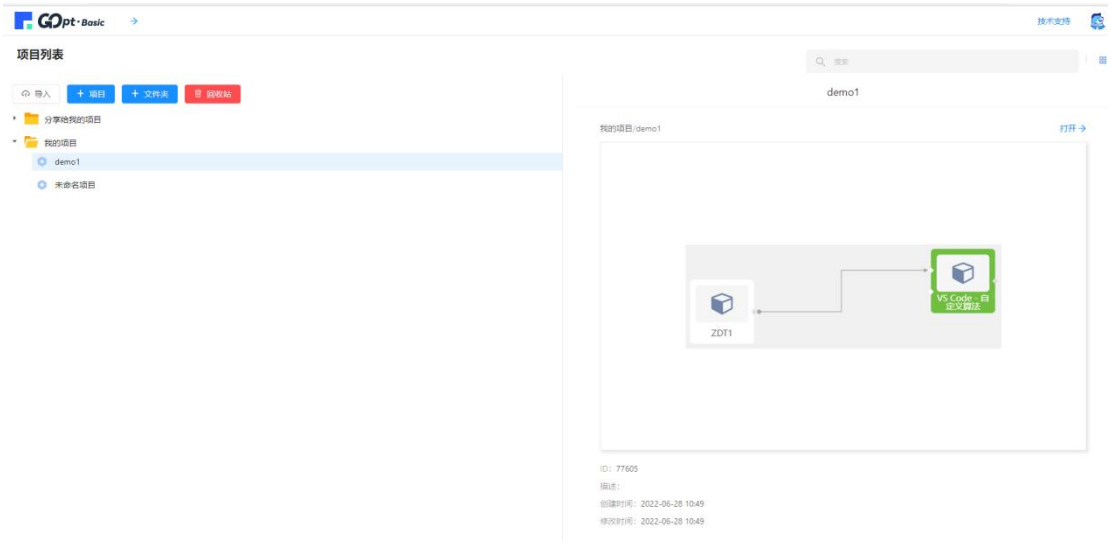


图 1

2）进入项目编辑界面并创建相应组件新建项目或打开已有项目并进入项目编辑界面。

从登录界面登陆后会进入到项目的管理界面，可以新建一个项目，或者打开一个已有项目并双击进入项目编辑界面，如下图所示。

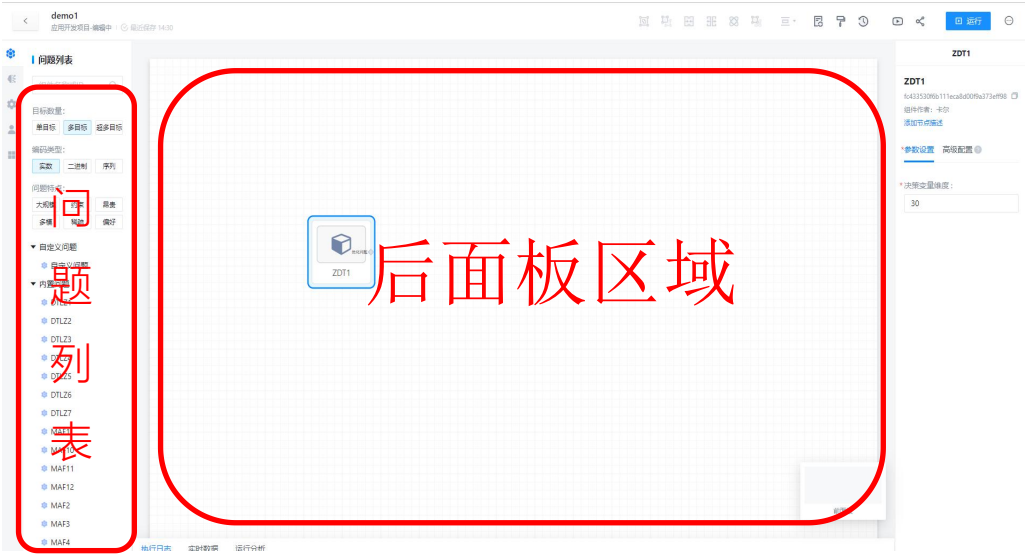


图 2

3）选择内置问题并拖拽至后面板区域。

从项目编辑界面左侧问题列表中筛选一个内置问题组件拖入到后面板区域中，选择的内置问题类型需要和自定义的算法类型匹配，内置问题的类型可以通过问题列表上方按钮选择。选定内置问题之后，点击最左侧的算法列表按钮（图 3A 区域），转到算法选择界面，并在自定义算法列表下找到 VS COde-自定义算法组件（图 3B 区域），将其拖入到后面板中，并将内置问题的输入接口与自定义算法组件的左上角输入接口相连，如图 3 所示：

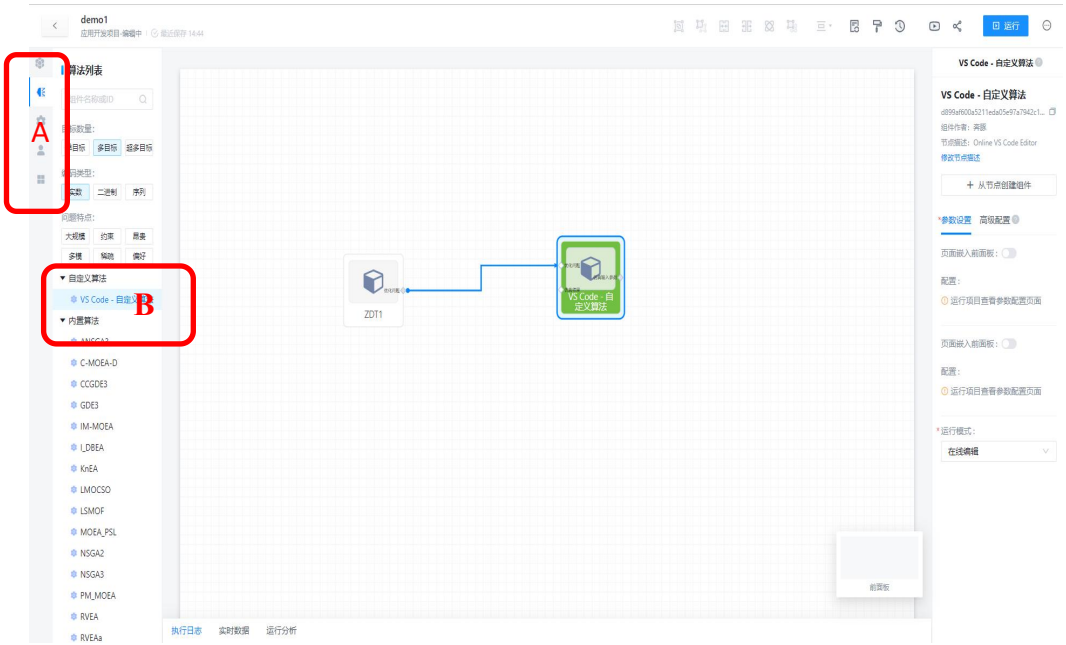


图 3

4）进入算法调试界面。

将组件相连之后，运行后即可进行调试，将鼠标置于右上方<运行>按钮处，会出现<运行/调试>按钮，选择调试，进入调试阶段。如图 4 所示：

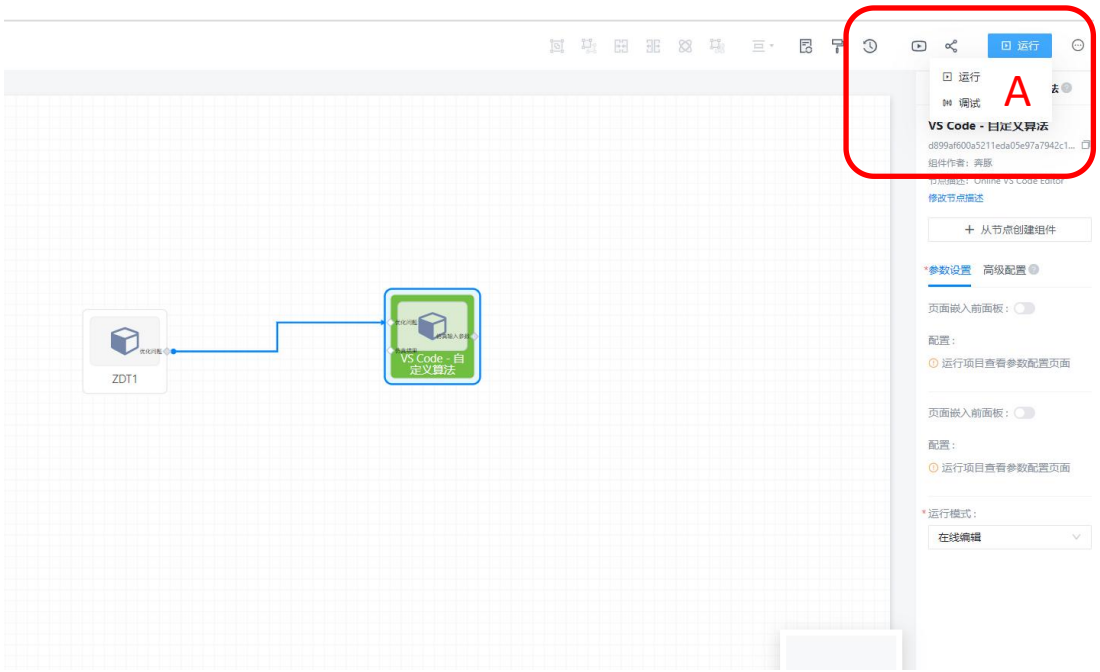


图 4

调试成功后，选中自定义算法组件，会出现如图 5 所示界面，可以点击图 5 中 A 区域的按钮以打开新的网页进行算法的编写。

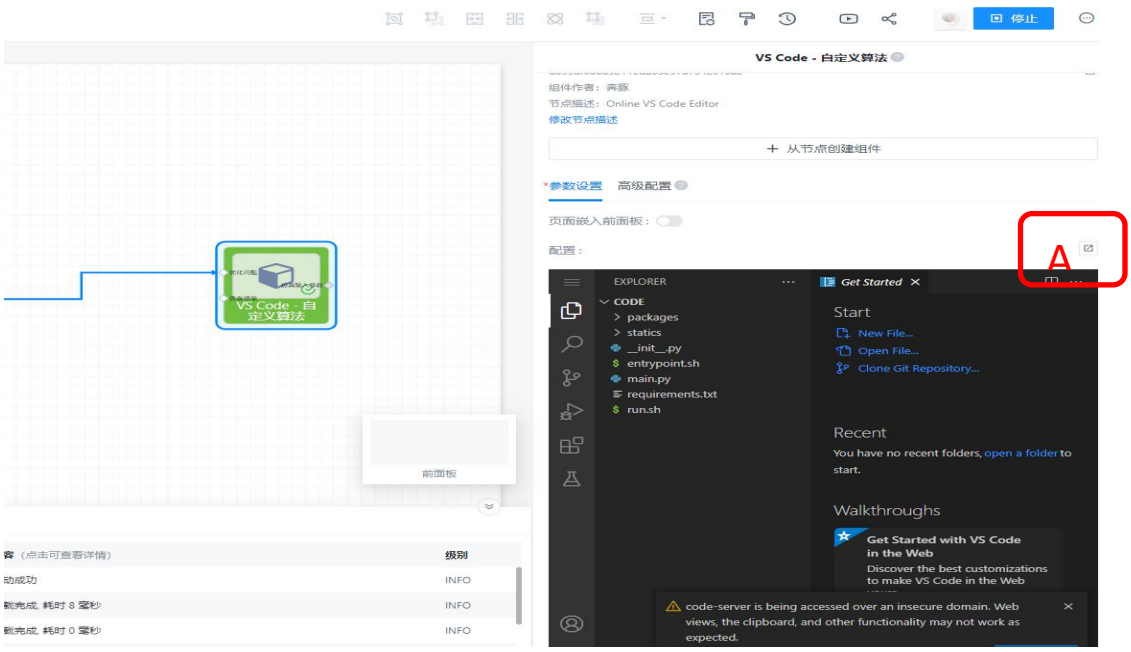


图 5

若在项目进入调试后选中自定义算法组件未出现图 5 所示界面，也属于正常现象，可以通过图 6 中 A 区域<刷新>按钮达到图 5 的展示效果。

图 6

5.2 算法实例

通过第二步最后点击拓展按钮打开新的界面，会进入到如下图所示的 VSCode 在线编辑界面，最左侧是 GOpt 提供的部分开源实例。

① 其中提供了一些公用函数放在 CODE/packages/platgo/utils 目录下，/utils 目录下的/selections 文件夹提供了几种环境选择的方法以供调用。

② CODE/packages/platgo/operator 目录下存放了部分常用的算子函数。

依次打开 CODE/packages/platgo/algorithms 文件夹，可以看到 NSGA2 算法的实例，可以按照此实例进行算法的编写。

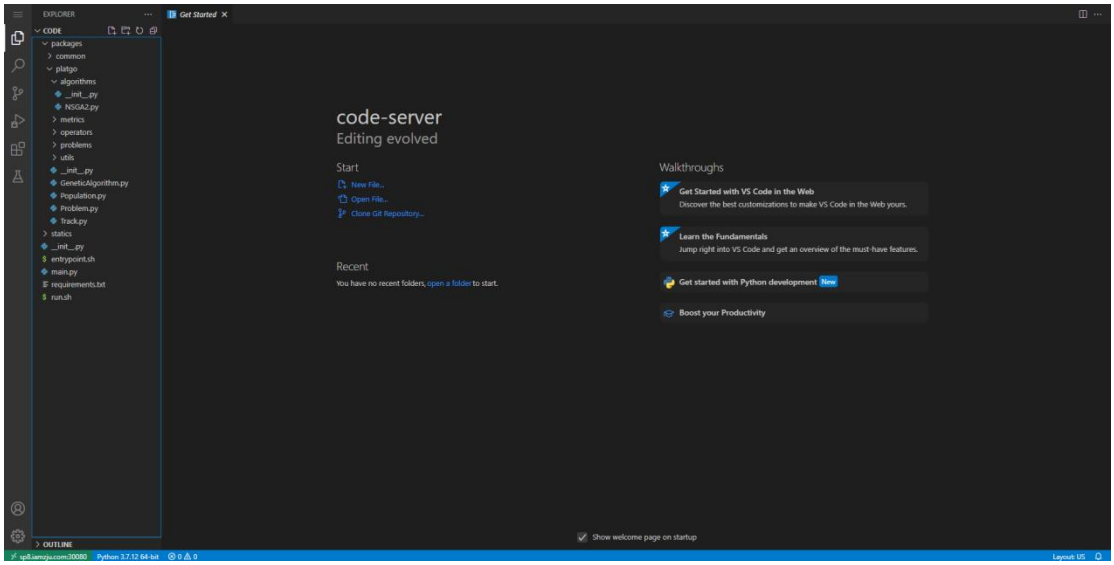


图 7

打开 NSGA2 的示例代码如下：

```
import numpy as np
from ...common.commons import AlgoMode
from .. import GeneticAlgorithm, utils, operators


class NSGA2(GeneticAlgorithm):
    type = {
        "n_obj": "multi",
        "encoding": {"real", "binary", "permutation"},
        "special": "constrained/none"
    }

    def __init__(
        self,
        pop_size,
        options,
        optimization_problem,
        control_cb,
        max_fe=10000,
```

```
        name="NSGA2",
        show_bar=False,
        sim_req_cb=None,
        algo_mode=AlgoMode.ACADEMIC,
        debug=False
):
        super(NSGA2, self).__init__(
                pop_size,
                options,
                optimization_problem,
                control_cb,
                max_fe=max_fe,
                name=name,
                show_bar=show_bar,
                sim_req_cb=sim_req_cb,
                algo_mode=algo_mode,
                debug=debug
        )

def run_algorithm(self):
        pop = self.problem.init_pop()
        self.cal_obj(pop)
        _, frontno, cd = self._environmental_selection(pop)

        while self.not_terminal(pop):
                matingpool = utils.tournament_selection(2, pop.pop_size, frontno, -cd)
                offspring = operators.OperatorGA(pop[matingpool], self.problem)
                self.cal_obj(offspring)  # 计算子代种群目标函数值
                temp_pop = offspring + pop  # 合并种群
                pop, frontno, cd = self._environmental_selection(temp_pop)
```

```
        return pop

    def _environmental_selection(self, pop):
        """
        The environmental selection of NSGA-II
        """
        # nd_sort
        if self.problem.n_constr > 0:
            frontno, maxfront = utils.nd_sort(
                pop.objv, pop.cv, self.problem.pop_size
            )  # noqa: E501
        else:
            frontno, maxfront = utils.nd_sort(pop.objv, self.problem.pop_size)
        next = frontno < maxfront
        # calculate the crowding distance
        cd = utils.crowding_distance(pop, frontno)

        # select the soltions in the last front based on their crowding distances  # noqa: E501
        last = np.argwhere(frontno == maxfront)
        rank = np.argsort(-cd[last], axis=0)
        next[
            last[rank[0: (self.problem.pop_size - np.sum(next))]]
        ] = True  # 先计算 next（也就是不是最大前沿上的）的总数，让 N-sum 即为需要在最后一前沿需要挑选的解
        # 此时选取最后前沿上 rank（即依据 cd 选取的）最好的解，再使其的 next 变为 Ture
        # pop for next generation
        pop = pop[next]  # 选取 next（即选中的解）中的解
        frontno = frontno[next]  # 选取这些解的前沿
```

```
        cd = cd[next]
        return pop, frontno, cd
```

第 1-3 行代码为导入相应的包或者函数，此处根据不同的算法导入不同的包。值得注意的是，在 GOpt 中，第 2 行代码的导入和第 3 行代码中的 GeneticAlgorithm 导入是必须的，其分别对应着 GOpt 的运行模式以及算法基类。

代码 4-34 行为 NSGA2 算法类以及其初始化函数。

第 4 行 NSGA2 继承了算法基类，用户自定义的算法同样也需要继承该类

第 5-9 行是算法的标签，该标签以字典的形式展现，用来定义算法可以解决的问题类型。在自定义算法中可以忽略此项。

第 10-22 行为 NSGA2 初始化所需要的参数列表，其中 self, pop_size, options, optimization_problem, control_cb, max_fe, name, show_bar, sim_req_cb, algo_mode, debug 中，self 为 Python 语法规定参数，其余均为算法基类构造器所需要的参数，上述参数中部分参数赋予了默认值，除去 pop_size,max_fe,name 用户可以进行修改以外，其余参数与 NSGA2 中一致即可。若用户自定义算法需要额外参数也可以在 init 函数参数列表添加。

第 23-34 行调用了算法基类的构造方法并传入相应的参数，自定义算法此处与 NSGA2 中一致即可。

第 35-45 行定义了算法的运行函数 run_algorithm，所有继承了算法基类的算法都需要实现该函数，该函数是函数运行的入口。

第 36 行初始化了种群，在一个具体的算法类中，算法通过 self.problem.init_pop()来创建一个新的种群，同时 init_pop()也可以传入一个整形数字来指定创建种群的大小，若不传参数，默认为 pop_size 的大小。

第 37 行通过调用 cal_obj()对创建好的种群进行了计算目标值操作，所有通过算子或者初始化方法产生的种群都需要计算目标值。计算目标值同时会将约束（若带约束）、越界修复（若需要）的工作一并完成。

第 38 行则是 NSGA2 算法自己逻辑操作。

第 39-45 行是算法迭代的部分，循环内部的逻辑需要用户自己实现。第 39 行通过调用 not_terminal()来判断算法是否到达停止条件，判断的标准为评价次数达到预定义的次数，参数为不断更新的种群。

第 45 行将最终得到的最优种群进行返回。

第 46-71 行为 NSGA2 算法环境选择函数，用户需要根据不同的算法逻辑进行编写。

5.3 算法运行与调试

在算法编辑界面 packages 包的同级目录下，有一个 main.py 文件，该文件可以实现 packages 包下算法的运行。

```
import suanpan
from suanpan.app import app
from suanpan.app.arguments import Json, Int


from packages.common.algo_front import algo_front, control_cb    # noqa
from packages.common.commons import AlgoMode
from packages.common.engineering_mode import sim_req_cb
from packages.platgo.algorithms import NSGA2



@app.input(Json(key="inputData1", alias="optimizationProblem"))
@app.input(Json(key="inputData2", alias="simulationResult"))
@app.param(Int(key="param1", alias="algoMode", default=0))
@app.param(Int(key="param2", alias="popSize", default=100))
@app.param(Int(key="param3", alias="maxFE", default=10000))
@app.output(Json(key="outputData1", alias="simulationParameter"))
def search_sqp(context):
    global algo_front
```

```
args = context.args


# 输入 1 问题
if args.optimizationProblem:
    algo_mode = AlgoMode(args.algoMode)
    if algo_mode is AlgoMode.ACADEMIC:
        evol_algo = NSGA2(
            args.popSize,
            {},
            args.optimizationProblem,
            control_cb,
            max_fe=args.maxFE,
            algo_mode=algo_mode)
    else:
        evol_algo = NSGA2(
            args.popSize,
            {},
            args.optimizationProblem,
            control_cb,
            max_fe=args.maxFE,
            sim_req_cb=sim_req_cb,
            algo_mode=algo_mode)
    evol_algo.start()
    algo_front.set_evol_algo(evol_algo)


# 输入 2 仿真结果返回
if args.simulationResult is not None:
    algo_front.add_simulation_result(args.simulationResult)
```

```
@app.afterInit
def init(context):
    global algo_front
    # 设置算法的仿真结果展示的弹窗
    algo_front.init_static()


if __name__ == "__main__":
    suanpan.run(app)
```

第 1-7 行行为导入该文件需要的包和类，其中 1-6 行是算盘所需要的，为必须导入的项，第 7 行导入的是该文件需要调用的算法类。

第 8-41 行定义了算盘调用算法的函数。

8-13 行：其中第 8、9、13 行为算法与算盘通讯所需参数，为必须传入的项，用户在编辑 main 文件时与上述代码一致即可。第 10、11、12 三行为 NSGA2 算法初始化所需传入的参数，如果在 NSGA2 算法类中这三行对应的三项参数赋予了默认值。并且在后续的算法持久化（将自定义算法生成一个算法组件）中不需要传入对应参数的值，可以不加。

14-41 行：14-41 行定义了函数的具体内容，该函数中仅需关注以下几点：15、16 行定义了一个全局变量并接收了 8-13 行中通过装饰器传入的参数。17-38 行进行了一个判断，18 行判断是否有测试问题对象传入，只有当有问题对象传入时才进行下面的操作。19 行获取了算法运行的模式，该参数来自文件的第 9 行，默认为 0 即可，当参数为 0 时，会走代码 21-27 行的判断部分（而不是走 else 部分，else 部分功能还处于完善状态）。21-27 代码对 NSGA2 算法进行了实例化，并传入了必要的参数。第 28-36 行代码可以忽略。第 37-41 行将算法的主函数作为一个线程启动，并将结果进行返回，自定义算法时该部分内容与此处一致即可。

第 42-46 行用于算法结果弹窗的展示，与文件中一致即可。

第 47-48 为主函数，启动算盘。

5.4 算法 debug

在算法以及 main 文件好并设置断点以后，可以通过点击图 8 中红色区域的 dubug 按钮进行调试。

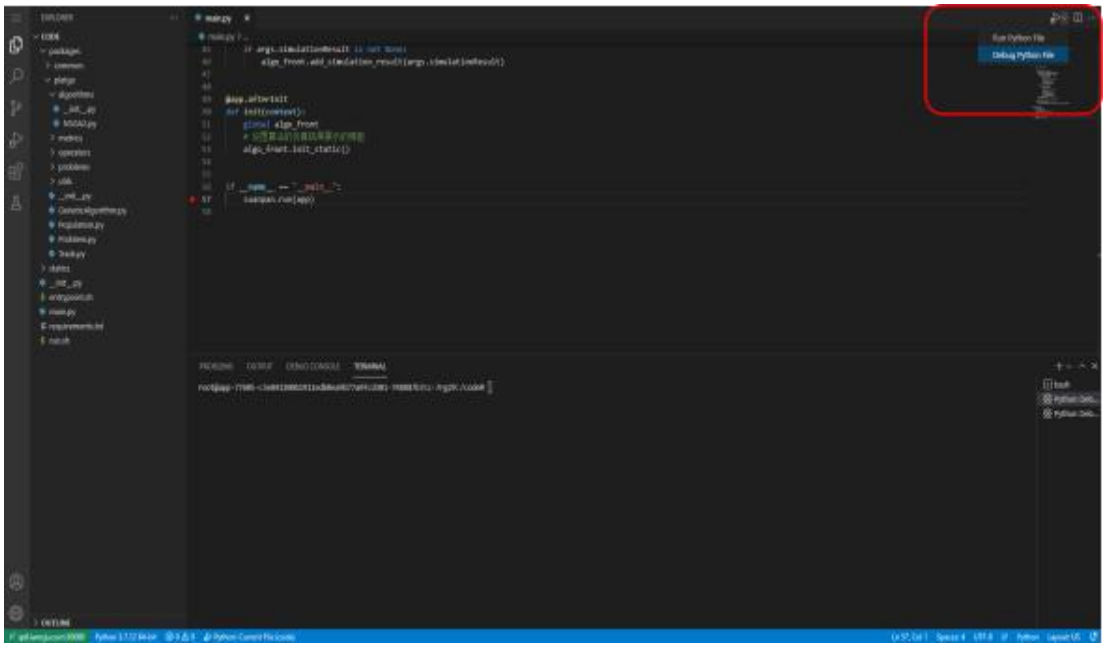


图 8

如果在点击 debug 按钮后出现如图 9 所示错误，可以在终端窗口输入 export GEVENT_SUPPORT=True

再此点击 debug 按钮进行调试。

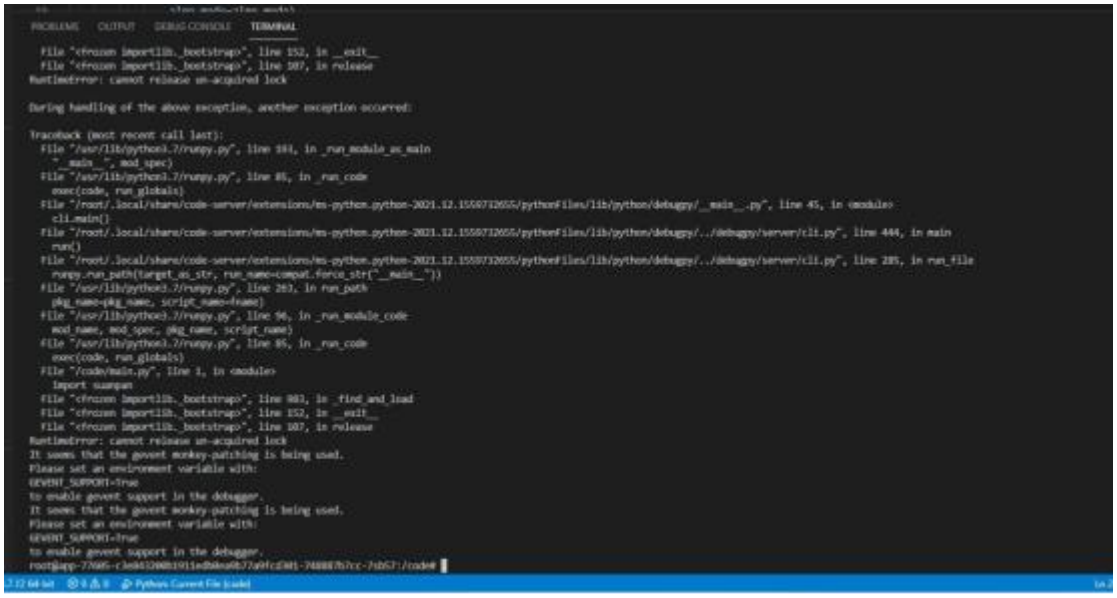


图 9

当 debug 运行图 8 中 57 行代码（即 suanpan.run(app)）后，需要打开后面板（如图 10 所示），点击图中红色部分刷新按钮后会显示出绘图展示（如图 11 所示），点击开始按钮后即可回到在线 IDE 进行调试。

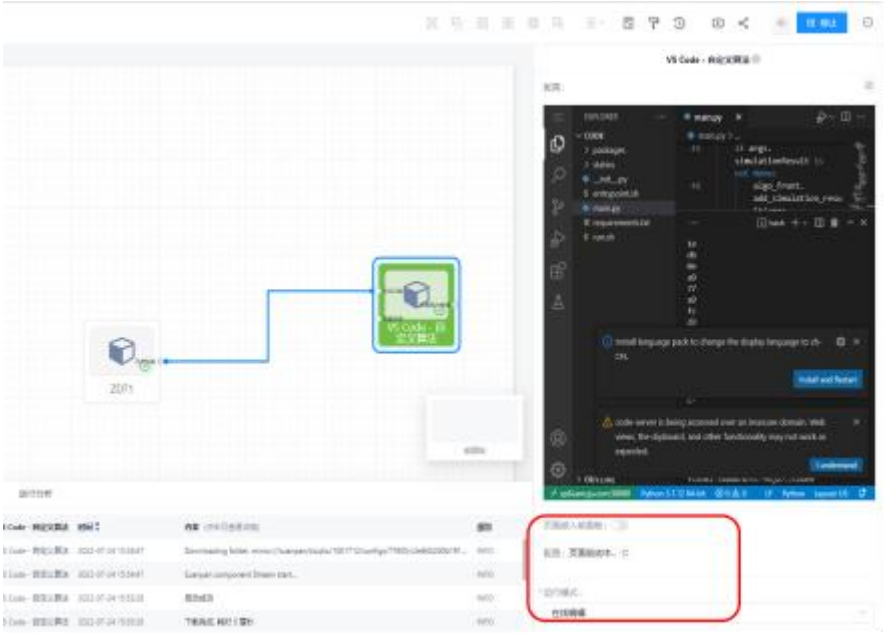


图 10

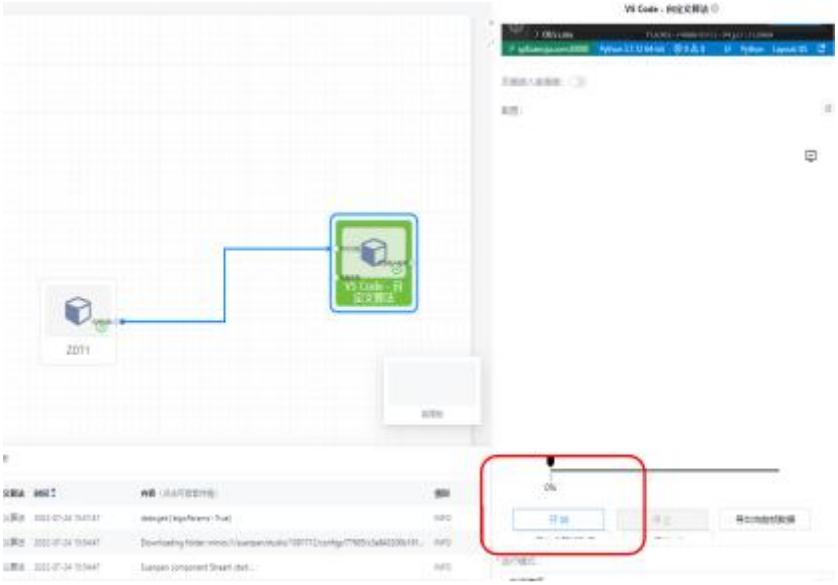


图 11

5.5 算法运行以及持久化（算法生成组件）

将算法定义完成后（即完成第四步后），在终端输入代码

./entrypoint.sh

即可运行算法。输入命令后转到后面板，点击图 10 所示按钮，得到图 11 所示界面，点击开始按钮即可运行算法。

算法持久化

在后面板中选中自定义算法组件，找到右边扩展栏中的“从节点创建组件”按钮，如图 12 所示：

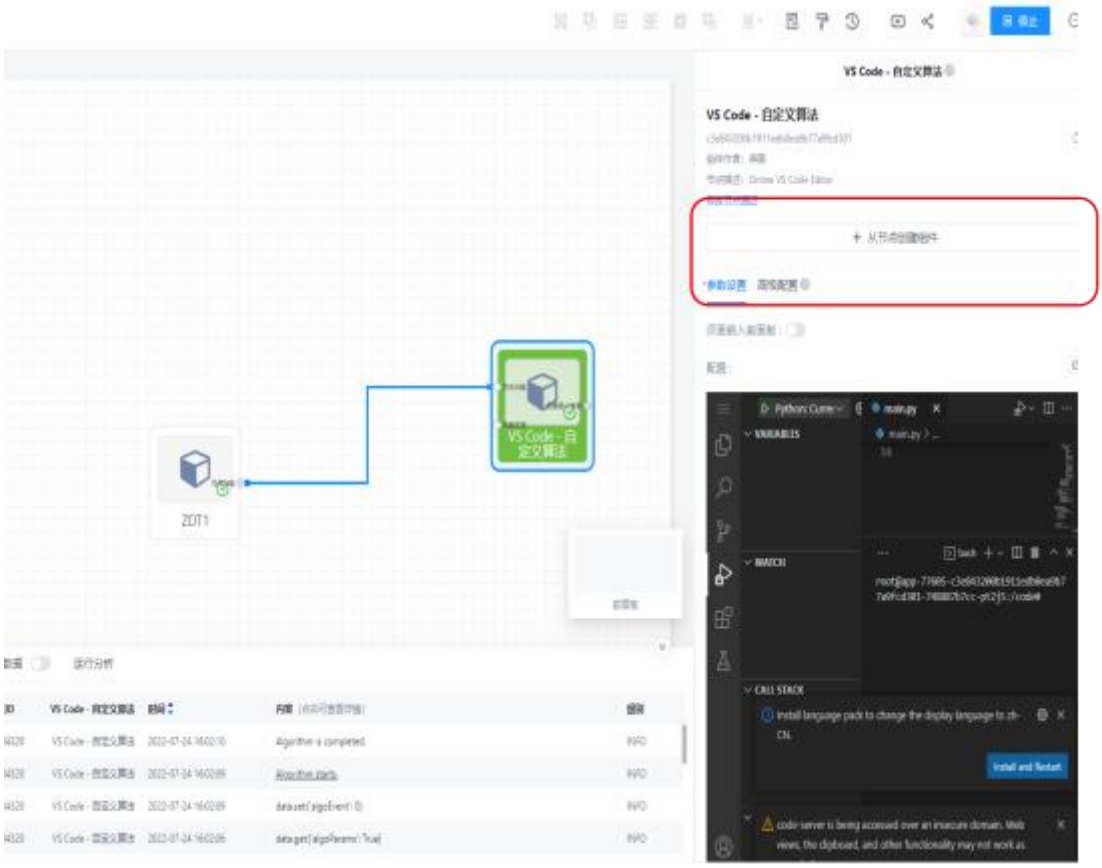


图 12

并给组件命名（如图 13）并点击创建按钮。

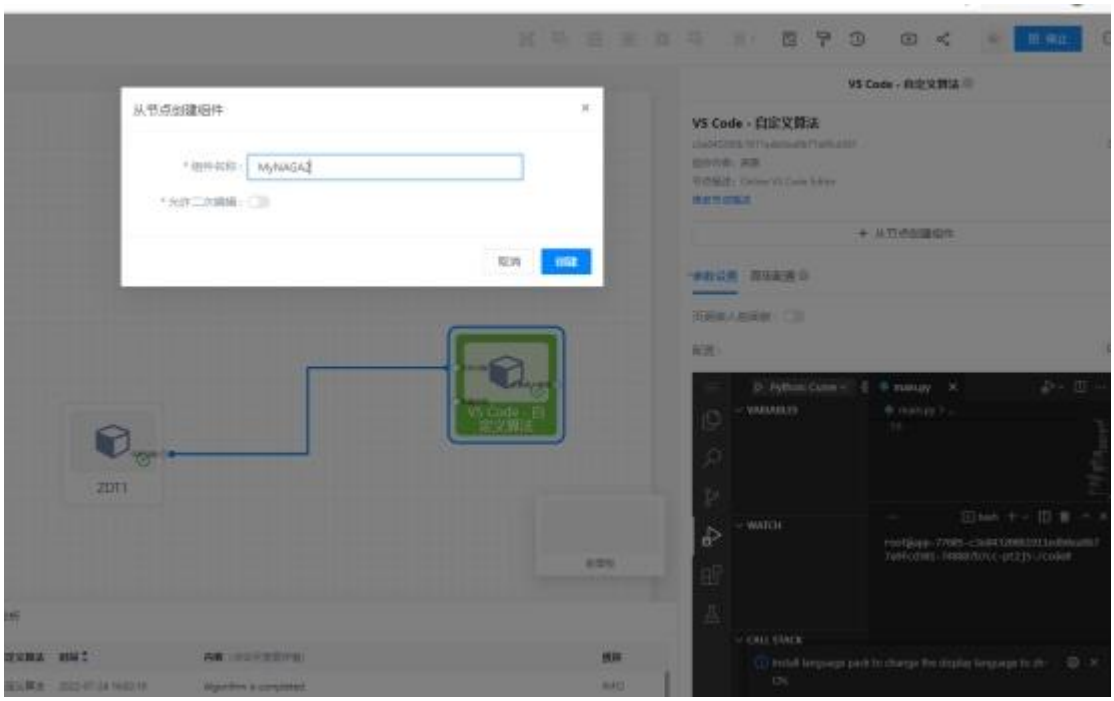


图 13

创建完成后会自动跳转到编辑组件界面，如图 14 所示，自定义组件需要在此处设置传递参数的控件。

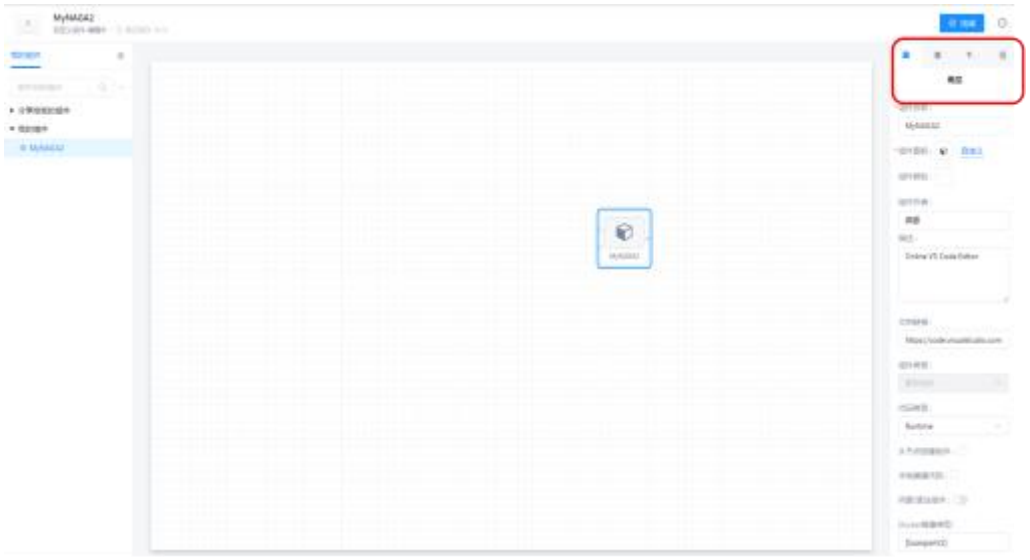


图 14

转到如图 14 所示界面后，需要点击图中红色矩形部分类似螺丝的按钮，进行控件编辑。编辑界面如图 15 所示，点击添加控件按钮会出现下来菜单以选择需要添加的控件类型。

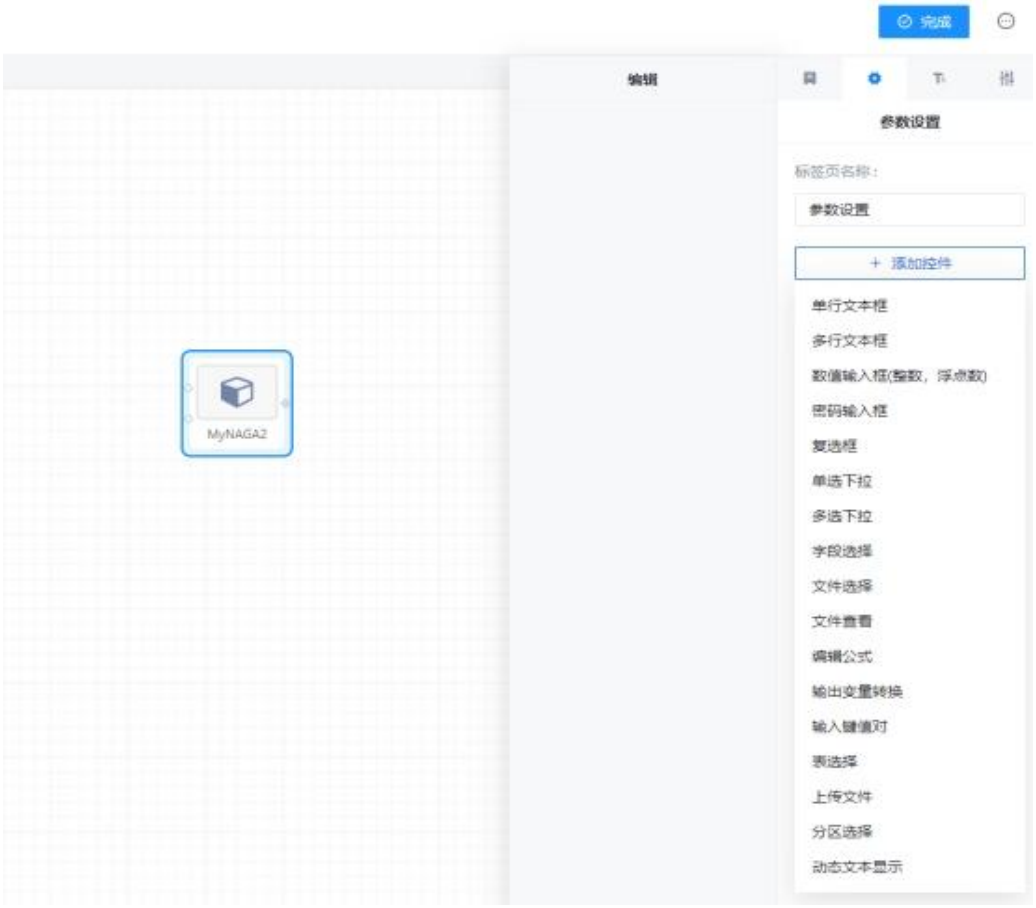


图 15

选定控件的标准为第四节代码中装饰器部分。

对于上述 8、9、13 行代码不需要配置，这是算盘内部会自动处理的参数。其余参数均需要配置。

```
@app.param(Int(key="param1", alias="algoMode", default=0))
@app.param(Int(key="param2", alias="popSize", default=100))
@app.param(Int(key="param3", alias="maxFE", default=10000))
```

下面为配置 popSize，以及 maxFE 参数的示例。

在图 15 的基础上，选择单行文本框，出现如下图所示界面（若仅出现“名称（唯一标识）”一行，在将这一行填入数据之后后续内容即可显示完全），将“名称（唯一标识）”设置为上述代码中 alias 的值，同时可以将占位提示文本改为“种群大小”，右侧的单行文本框内可以填入默认值。

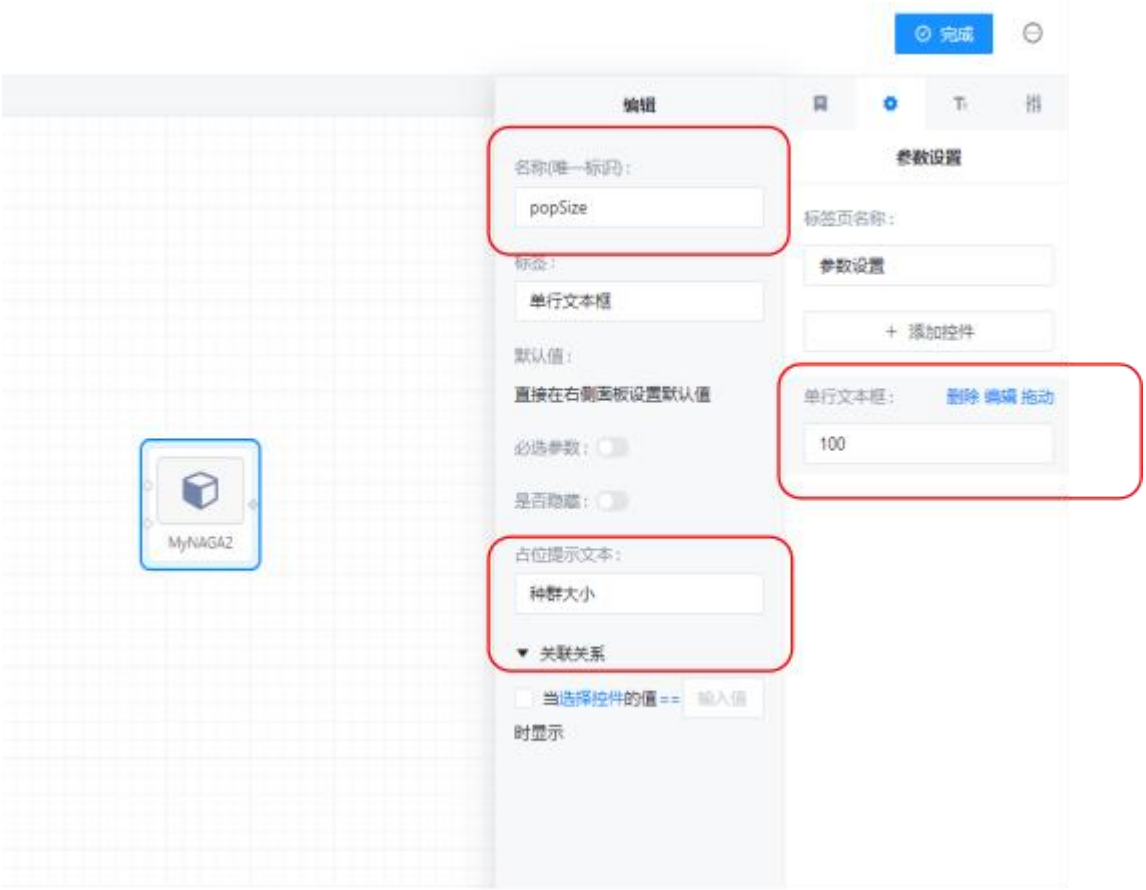


图 16

popSize 参数设置完成后，继续点击图 15 所示的添加控件按钮，为 maxFE 添加输入框，同样选择单行文本框。步骤与添加 popSize 一致，如图 17：

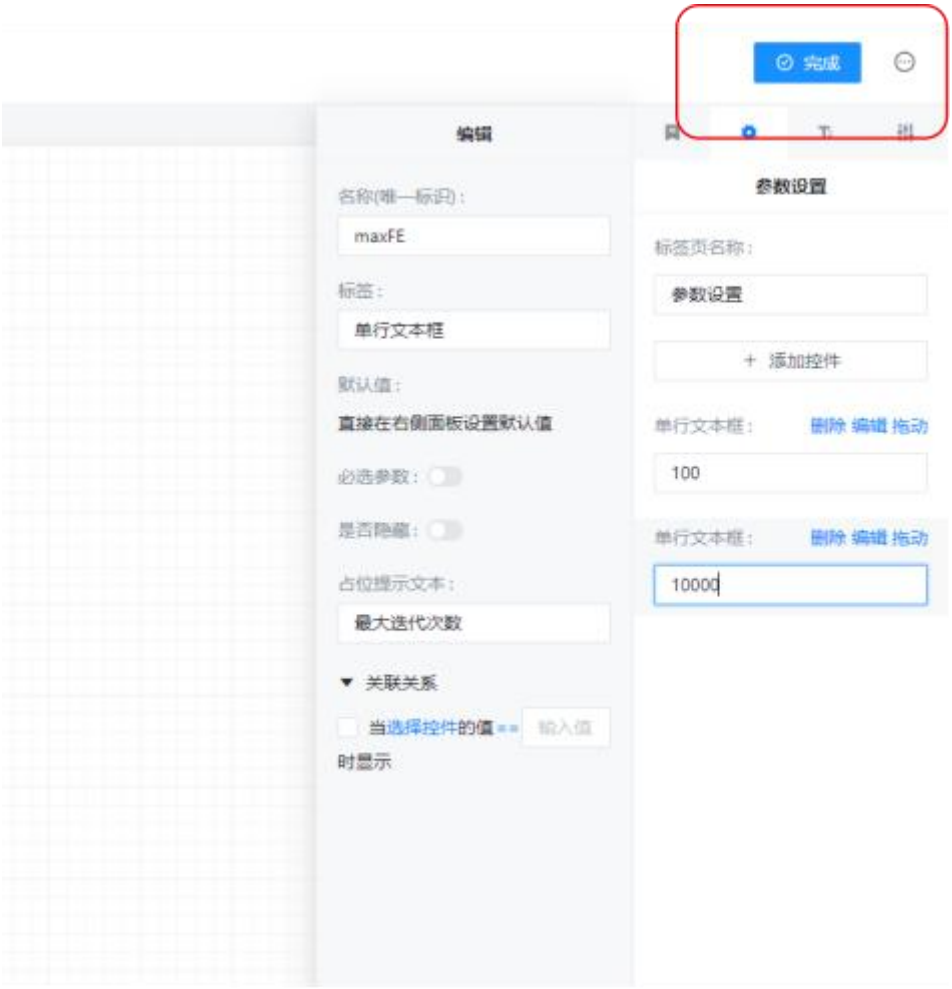


图 17

将所有参数设置完毕后点击图 17 中红色矩形部分的完成按钮。完成后该组件可以在我的组件列表找到。如果自定义算法需要额外的参数需要在第四部代码的装饰器部分进行添加，并在组件上创建控件以进行输入，如果在装饰器中赋予了默认值或者算法的构造器函数的形参列表赋予了默认值，在组件的控件创建部分可以不进行创建，组件将会直接按照默认值进行运行。

第六章 扩展 GOpt

6.1 算法类

每个算法需要被定义为 GeneticAlgorithm 类的子类并保存在 packages/platgo/algorithms 文件夹中。算法类包含的属性与方法如下：

属性	赋值方式	描述
pop_size	用户	种群大小
max_fe	用户	最大评价次数
name	用户	算法名称
algo_mode	用户	算法运行模式，包括学术模式和工程模式（目前工程模式尚在开发）
其余属性	平台	涉及与算盘通讯、显示所需参数
方法	是否可重新定义	描述
run_algorithm	必须	算法主体部分
not_terminal	不可	算法每次迭代前调用的函数
_draw	不可	用于本地绘制图像
_validate	不可	用于算盘上自定义问题组件验证问题
cal_obj	不可	通过调用 Problem 类求解问题
init_problem	不可	用于初始化问题
其余方法	不可	涉及与算盘通讯、显示所需方法

每个算法需要继承 GeneticAlgorithm 类并重新定义方法 run_algorithm()。例如章节 5.2 算法实例中 NSGA2。算法中调用了许多 packages/platgo/operators 和 packages/platgo/utils 文件夹下的公共函数。具体如下：

函数名	描述
OperatorDE	差分进化算子

OperatorFEP	进化规划算子
OperatorGA	遗传算子
OperatorGAhalf	遗传算子（仅产生前一半的子代）
OperatorPSO	粒子群优化算子
DAE	Feedforward neural network
RBM	Restricted Boltzmann Machine
CCDE	Differential evolution operator used in CCGDE3
naive_selection	按目标值大小顺序选择
random_selection	随机选择
roulette_wheel_selection	轮盘赌选择
tournament_selection	联赛选择
crowding_distance	计算解的拥挤距离（用于多目标优化）
fitness_single	计算解的适应度（用于单目标优化）
nd_sort	非支配排序
uniform_point	产生均匀分布的参考点

6.2 问题类

每个问题需要被定义在 Problem 类的子类并保存在 packages/platgo/problems 文件夹中。

目前用户尚只能在自定义问题组件中自定义问题。详见章节四——自定义问题组件，创建自定义问题。

问题类包含的属性与方法如下：

属性	赋值方式	描述
pop_size	用户	求解该问题的算法的种群大小
name	用户	算法名称
n_obj	用户和问题类	问题的目标数
n_var	用户和问题类	问题的变量数
n_constr	用户和问题类	问题约束的数量
max_fe	用户	求解该问题可使用的最

		大评价次数
encoding	问题类	问题的编码方式
lower	问题类	决策变量的下界
upper	问题类	决策变量的上界
variables	用户	用于函数计算的中间变量
initFcn	用户	用于初始化种群
decFcn	用户	用于修复违反上下界范围的决策变量
objFcn	用户	用于计算目标函数值
conFcn	用户	用于计算目标约束值
gradFcn	用户	用于计算目标函数的梯度
conGradFcn	用户	用于计算约束函数的梯度
data_sets	用户	用于接收用户上传的数据集
type	用户	用于定义问题标签
方法	是否可重定义	描述
init_pop（）	可以	初始化一个种群
compute()	可以	用于计算问题目标值、约束值、梯度值、修复决策变量
_validate（）	不可以	用于验证用户定义问题是否合法
其余方法	不可以	涉及与算盘通讯、显示所需方法

每个问题需要继承 Problem 类并在__init__（）内初始化问题的各个参数，包括函数的计算。也可以通过重写 compute（）函数计算各个函数。例如 ZDT1.py 的代码为：

```
from ....Problem import Problem


class ZDT1(Problem):
    type = {
```

```
    "n_obj": "multi",
    "encoding": "real",
    "special": {"large/none", "expensive/none"},
}

def __init__(self, in_optimization_problem={}) -> None:
    optimization_problem = {
        "name": "ZDT1",
        "encoding": "real",
        "n_var": 30,
        "lower": "0",
        "upper": "1",
        "n_obj": 2,
        "variables": {
            "g": "1 + 9 * mean(x[1:])",
            "h": "1 - sqrt(x[0] / g)"
        },
        "initFcn": [],
        "decFcn": [],
        "objFcn": ["x[0]",
            "g * h"],
        "conFcn": ["0"]
    }
    optimization_problem.update(in_optimization_problem)
    super(ZDT1, self).__init__(optimization_problem)
```

第 1 行代码为导入相应的包或者函数，此处 Problem 导入是必须的，对应着 GOpt 的问题基类。

第 2 行 ZDT1 继承了问题基类。

第 3-7 行是问题的标签，该标签以字典的形式展现，用来定义算法可以解决的问题类型。用户在自定义问题组件中有相应的功能为自定义问题添加便签。

第 8-25 行问题初始化，分别定义了问题的名字、编码方式、决策变量长度、决策变量下界、决策变量上界、目标数、函数计算时调用的中间参数、种群初始化函数、决策变量修复函数、目标函数、约束函数。其中种群初始化函数、决策变量修复函数等设空，则使用问题基类中的默认函数。若用户需要额外参数也可以在 init 函数中添加。

第 26 行用于接收来自算盘上用户输入的参数值，并更新。

第 27 行调用了问题基类的构造方法并传入相应参数。

除了以上代码外，对于问题各个函数的计算，若用户觉得在 init 中计算繁琐，也可重写 compute（）函数用于各个函数的计算。

6.3 种群类

一个 Population 类的对象表示一个个体（即一个解），一组 Population 类的对象表示一个种群。个体类包含的属性和方法如下：

属性	赋值方式	描述
decs	用户	解的决策向量
objv	用户	解的目标值
vel	用户	解的约束违反值
cv	用户	解的额外属性值（例如速度）
方法	描述	
copy（）	返回一个种群的副本	
__getitem__（）	种群切片。根据下标选择部分个体生成新的种群	
__setitem__（）	为种群内的部分个体赋值，支持多对一	
__add__（）	合并种群，不更改原种群，返回新种群	

例如以下代码产生一个具有十个解、每个解长度为 20 的种群

```
decs = np.random.random(size=(10, 20))
Population(decs=decs)
```

此时种群只有决策变量值，之后可通过调用 GeneticAlgorithm 类中的 cal_obj（）函数计算种群的各个函数值。