音乐与数学：遗传算法研究报告

物理学院 江泽辉 陈鹏

May 2023

1 引言

我们的小组编号为 4-8，分别由两位来自物理学院的同学江泽辉和陈鹏、以及两位来自元培学院的同学郝炜天和王雍淐组成。

在这次小组作业中，我们选择的是第四题：《机器作曲 · 遗传算法》。其中郝炜天负责了对参考文献 [1] 中构建的遗传算法的基本复现，并加入了 crossover 这种操作函数；江泽辉在郝炜天代码的基础上，增加了对于旋律变换的操作函数——移调、倒影和逆行变换；江泽辉和陈鹏在研读参考文献 [1] 的基础上，总结了文献中没有考虑全面的点，并仔细探讨了适应度函数中应该考虑的因素，提出了一些备选的待实现的方案；陈鹏在郝炜天的代码基础上进一步完善了适应度函数、以及输出音频接口的实现；王雍淐提供了指导意见和一些可供参考的音乐片段。最后的报告由组长江泽辉和陈鹏共同完成。

2 文献中遗传算法的建立

我们根据参考文献 [1] 第二章给出的方法，在 Python 上建立了基本的遗传算法。该算法包含了产生初始种群、三种变异 (Mutation) 操作，和增添的旋律变换——移调、倒影和逆行。由于仅仅是一个 “草稿”，在这个代码中我们将适应度函数设定为向量之间的距离。可以认为，这是最简化的适应度函数了。在本文的第三部分中，我们会对适应度函数进行细致地修改，以产生更好的音乐片段。

在本程序中，我们默认音乐片段的拍号为 4/4 拍，并采取文献中的编码方式：在两个八度之内，以八分音符为最小时间单位；将根音编号为 1，冠音编号为 14；由于在常规的音乐片段中，总会出现延长音和空拍，我们将 15 作为将前一个音延长一个八分音符时值的编码、将 0 作为八分休止符的编码。

在实现的过程中，将小星星作为参考片段 (Reference Melody)：

```
reference_melody =
[1,1,5,5,6,6,5,15,4,4,3,3,2,2,1,15,5,5,4,4,3,3,2,15,5,5,4,4,3,3,2,15,
1,1,5,5,6,6,5,15,4,4,3,3,2,2,1,15,5,5,4,4,3,3,2,15,5,5,4,4,3,3,2,15]
```

参考片段指明了遗传算法中产生种群的进化方向。

2.1 产生初始种群

我们采用 create_answer() 函数产生在一定的乐音体系中、一定长度的、随机的种群。即，我们选取初始种群方式是利用 Python 的 random 库随机生成。

2.2 具体的函数模块

我们根据主程序的逻辑顺序，依次介绍在实现遗传算法的过程中定义的函数。

2.2.1 初始种群的生成：create_answer(n)

利用 Python 中的 random 库实现随机生成初始种群，每个个体的长度为 64，随机数落在 0-15 之间（代表选取的音级落在两个八度之间）；自变量 n 代表产生个体的个数。

```
import random
import numpy as np
# 随机生成n个旋律（如果有种子的话就不需要这个函数了）
# 按文献里的表示方法，将所有旋律限制在某个特定大调（例如C大调）的两个八度之内
# 不考虑升降号
def create_answer(n):
    result = []
    for i in range(n):
        result.append(np.random.choice(range(0,16), 64))
    return result
```

2.2.2 计算适应度：fitness_evaluation(mutated_population)

```
# 将适应度设定为某段旋律与reference的距离（也就是向量之差的模长）的倒数
def fitness_evaluation(mutated_population):
    fitness_value = []
    for item in mutated_population:
        new_melody = np.array(item)
        error = np.linalg.norm(new_melody-ref_melody)
        fitness_value.append(1/(error+1))
    return fitness_value
```

2.2.3 选择函数：selection(input)

```
# 定义被选择的概率，选中的概率与旋律适应度有关
def selection(input):
    fitness_value = fitness_evaluation(input)
    # 归一化，得到的就是每段旋律被抽到的概率，适应度越高越容易被抽到
    fitness_one = [item/sum(fitness_value) for item in fitness_value]
    for i in range(1, len(fitness_one)):
        fitness_one[i] += fitness_one[i-1]
        # 变异时从0到1均匀分布抽取数，抽出的数所在的区间代表需要变异的旋律
        # 例如，抽出0.45，在fitness_one[3]和fitness_one[4]之间，那么就让下标为4的旋律变异
    return fitness_one
```

2.2.4 交叉操作：crossover(input, fitness_input, n)

```
def crossover(input, fitness_input, n):
    output = []
    for i in range(n):
        # 首先选取两段旋律，注意不能抽到一样的
        rand = random.uniform(0, 1)
        index1 = 0
```

```
        for j in range(0, len(input)):
            if fitness_input[j] > rand:
                index1 = j
                break
        rand = random.uniform(0, 1)
        index2 = 0
        for j in range(0, len(input)):
            if fitness_input[j] > rand:
                index2 = j
                break
        while index2 == index1:
            rand  = random.uniform(0, 1)
            for j in range(0, len(input)):
                if fitness_input[j] > rand:
                    index2 = j
                    break
        tmp1 = input[index1].copy()
        tmp2 = input[index2].copy()

        # 然后选取两个下标x和y，交换[x+1, y]之间的部分
        x = random.randint(0, 62)
        y = random.randint(x + 1, 63)
        tmp1_c = list(tmp1[:x+1]) + list(tmp2[x+1:y+1]) + list(tmp1[y+1:])
        tmp2_c = list(tmp2[:x+1]) + list(tmp1[x+1:y+1]) + list(tmp2[y+1:])
        output.append(tmp1_c)
        output.append(tmp2_c)

    print("crossed-over")
    return output
```

2.2.5 变异操作 1：变化一个八度 change_for_octave(input, fitness_input, n)

```
# Mutation 1: Changing tone for an octave.
# 将某个音变化一个八度，体现在数字上就是+7或-7，这里需要注意不能选到0和15
def change_for_octave(input, fitness_input, n):
    output = []
    for i in range(n):
        rand = random.uniform(0, 1)
        index = 0
        for j in range(0, len(input)):
            if fitness_input[j] > rand:
                index = j
                break

        rand_index = random.randint(0, 63)
        while input[index][rand_index] == 0 or input[index][rand_index] == 15:
            rand_index = random.randint(0, 63)

        tmp = input[index].copy() # 注意深拷贝

        if tmp[rand_index] > 7:
            tmp[rand_index] -= 7
        else:
            tmp[rand_index] += 7

        output.append(tmp)
```

```
    print("changed for octave")
    return output
```

2.2.6 变异操作 2：更改一个音符 change_one_note(input, fitness_input, n)

```
# Mutation 2: Changing one tone.
# 随机改变数组中的某个数，这里不必考虑那个被改变的数是不是15，但是第一个数不能是15
def change_one_note(input, fitness_input, n):
    output = []
    for i in range(n):
        rand = random.uniform(0, 1)
        index = 0
        for j in range(0, len(input)):
            if fitness_input[j] > rand:
                index = j
                break

        rand_index = random.randint(0, 63)
        #print(rand, index, rand_index, input[index])

        tmp = input[index].copy()
        tmp[rand_index] = random.randint(0, 15)
        while rand_index == 0 and tmp[rand_index] == 15:
            tmp[rand_index] = random.randint(0, 15)
        output.append(tmp)

    print("changed one note")
    return output
```

2.2.7 变异操作 3：交换两个连续的音符 swap_two_notes(input, fitness_input, n)

```
# Mutation 3: Swapping two consecutive notes.
# 交换两个音符，既然是音符，就要考虑不能抽到15
def swap_two_notes(input, fitness_input, n):
    output = []
    for i in range(n):
        rand = random.uniform(0, 1)
        index = 0
        for j in range(1, len(input)):
            if fitness_input[j] > rand:
                index = j
                break

        # 选中一个音符，要跟它前面的音符交换
        snd_index = random.randint(1, 63)
        while input[index][snd_index] == 15:
            snd_index = random.randint(1, 63)
        fst_index = snd_index - 1
        fst_len = 1
        while fst_index == 15:
            fst_index -= 1
            fst_len += 1
        snd_len = 1
```

```
        for j in range(snd_index+1, len(input[index])):
            if input[index][j] != 15:
                break
            snd_len += 1

        tmp = input[index].copy()

        for j in range(fst_len + snd_len):
            tmp[fst_index + j] = 15
        tmp[fst_index] = tmp[snd_index]
        tmp[fst_index + snd_len] = tmp[fst_index]
        output.append(tmp)
    print("swapped two notes")
    return output
```

2.2.8 旋律变换 1：移调 transposition_exact(input, fitness_input, n):

```
# Melody transformation: transposition
# 移调变换：定义严格移调，不考虑调性移调
# 严格移调：将每个音级升高or降低相同的音级
def transposition_exact(input, fitness_input, n):
    output = []
    for i in range(n):
        #随机选取一个个体，记录下其下标
        rand = random.uniform(0, 1)
        index = 0
        for j in range(1, len(input)):
            if fitness_input[j] > rand:
                index = j
                break

        #利用随机数抽签决定：升高或者降低一个音级
        rand = random.uniform(0,1)
        tmp = input[index].copy()
        if rand >= 0.5: #升高一个音级
            for j in range(0,len(input[index])):
                if input[index][j] == 0 or input[index][j] == 15:
                    continue
                #注意边界的音级存在越界
                if input[index][j] == 14:
                    tmp[j] = 1
                else:
                    tmp[j] += 1
        else:            #降低一个音级
            for j in range(0,len(input[index])):
                if input[index][j] == 0 or input[index][j] == 15:
                    continue
                #注意边界的音级存在越界
                if input[index][j] == 1:
                    tmp[j] = 14
                else:
                    tmp[j] -= 1
        output.append(tmp)

    print("transposed")
    return output
```

2.2.9 旋律变换 2：倒影 inversion(input, fitness_input, n)

```
# Melody transformation: inversion
# 倒影变换：把旋律中的上升音程用相同音级差的下降音程代替；反之亦然
def inversion(input, fitness_input, n):
    output = []
    #随机选取一个个体，记录下其下标
    for i in range(n):
        rand = random.uniform(0, 1)
        index = 0
        for j in range(1, len(input)):
            if fitness_input[j] > rand:
                index = j
                break
        #以7为中心，做倒影变换：7-7,6-8,x - (14-x)
        tmp = input[index].copy()
        for j in range(0,len(input[index])):
            if input[index][j] == 0 or input[index][j] == 15:
                continue
            else:
                tmp[j] = 14 - input[index][j]
        output.append(tmp)
    print("inversed")
    return output
```

2.2.10 旋律变换 3：逆行 retrograde(input, fitness_input, n)

```
# Melody transformation: retrograde
# 逆行变换：把一段旋律“从尾到头”重读一遍
# 以小节为单位变换；需要注意的是0 和 15 作为小节末尾的处理
def retrograde(input, fitness_input, n):
    output = []
    #随机选取一个个体，记录下其下标
    for i in range(n):
        #index对应个体的下标，index1对应小节的下标
        rand = random.uniform(0, 1)
        index = 0
        for j in range(1, len(input)):
            if fitness_input[j] > rand:
                index = j
                break
        # 利用随机数抽取一个小节
        rand1 = random.uniform(0,1)
        index1 = 0
        for j in range(1, len(input)):
            if fitness_input[j] > rand1:
                index1 = j
                break
        # 我们考虑的个体长8小节，每小节8个八分音符
        temp = input[index].copy()

        if 0 <= index1 < 0.125:
```

```
        # 处理第一个小节中的音符
        temp_bar = list(temp[0 : 8])
        #注意最后一个音符的处理
        if temp_bar[7] == 0 or temp_bar[7] == 15:
            for j in range (7):
                temp[j] = temp_bar[6 - j]
        else:
            for j in range (8):
                temp[j] = temp_bar[7 - j]
        output.append(temp)

    elif 0.125 <= index1 < 0.25:
        # 处理第二个小节中的音符
        temp_bar = list(temp[8 : 16])
        if temp_bar[15] == 0 or temp_bar[15] == 15:
            for j in range (8,15):
                temp[j] = temp_bar[14 - j]
        else:
            for j in range (8,16):
                temp[j] = temp_bar[15 - j]
        output.append(temp)

    elif 0.25 <= index1 < 0.375:
        # 处理第三个小节中的音符
        temp_bar = list(temp[16 : 24])
        if temp_bar[23] == 0 or temp_bar[23] == 15:
            for j in range (16,23):
                temp[j] = temp_bar[22 - j]
        else:
            for j in range (16,24):
                temp[j] = temp_bar[23 - j]
        output.append(temp)

    elif 0.375 <= index1 < 0.5:
        # 处理第四个小节中的音符
        temp_bar = list(temp[24 : 32])
        if temp_bar[31] == 0 or temp_bar[31] == 15:
            for j in range (24,31):
                temp[j] = temp_bar[30 - j]
        else:
            for j in range (24,32):
                temp[j] = temp_bar[31 - j]
        output.append(temp)

    elif 0.5 <= index1 < 0.625:
        # 处理第五个小节中的音符
        temp_bar = list(temp[32 : 40])
        if temp_bar[39] == 0 or temp_bar[39] == 15:
            for j in range (32,39):
                temp[j] = temp_bar[38 - j]
        else:
            for j in range (32,40):
                temp[j] = temp_bar[39 - j]
        output.append(temp)

    elif 0.625 <= index1 < 0.75:
        # 处理第六个小节中的音符
```

```
            temp_bar = list(temp[40 : 48])
            if temp_bar[47] == 0 or temp_bar[47] == 15:
                for j in range (40,47):
                    temp[j] = temp_bar[46 - j]
            else:
                for j in range (40,48):
                    temp[j] = temp_bar[47 - j]
            output.append(temp)

        elif 0.75 <= index1 < 0.875:
            # 处理第七个小节中的音符
            temp_bar = list(temp[48 : 56])
            if temp_bar[55] == 0 or temp_bar[55] == 15:
                for j in range (48,55):
                    temp[j] = temp_bar[54 - j]
            else:
                for j in range (48,56):
                    temp[j] = temp_bar[55 - j]
            output.append(temp)

        elif 0.875 <= index1 < 1:
            # 处理第八个小节中的音符
            temp_bar = list(temp[56 : 64])
            if temp_bar[63] == 0 or temp_bar[63] == 15:
                for j in range (56,63):
                    temp[j] = temp_bar[62 - j]
            else:
                for j in range (56,64):
                    temp[j] = temp_bar[63 - j]
            output.append(temp)

    print("reversed")
    return output
```

2.3 主程序

在主程序中，我们设定参考旋律 (Reference Melody) 为小星星，长 8 小节；设置迭代次数 t 为 100，通过随机生成的方式产生初始种群。在每次迭代的过程中，依次输出当前的迭代次数“Current: t”，种群大小“length: len”，以及在每次操作之后操作的执行情况，如输出“Crossed-over”，“Changed for octave” 等等。最后，在每次迭代后输出当前迭代的结果。

```
reference_melody = [1,1,5,5,6,6,5,15,4,4,3,3,2,2,1,15,5,5,4,4,3,3,2,15,5,5,4,4,3,3,2,15,
1,1,5,5,6,6,5,15,4,4,3,3,2,2,1,15,5,5,4,4,3,3,2,15,5,5,4,4,3,3,2,15]
ref_melody = np.array(reference_melody)
initial_population = create_answer(10)
input = initial_population

for t in range(100):
    print("Current: ", t)
    print("length: ", len(input))
    fitness_input = selection(input)

    output_crossover = crossover(input, fitness_input, 50)
    input = list(input) + output_crossover
    print("length: ", len(input))
```

```
    fitness_input = selection(input)

    output_octave = change_for_octave(input, fitness_input, 100)
    output_note = change_one_note(input, fitness_input, 100)
    output_swap = swap_two_notes(input, fitness_input, 100)
    output_transposition = transposition_exact(input, fitness_input, 100)
    output_inversion = inversion(input, fitness_input, 100)
    output_retrograde = retrograde(input, fitness_input, 100)

    mutated_population = list(output_octave) + list(output_note) + list(output_swap)
    + list(output_transposition) + list(output_inversion) + list(output_retrograde)

    fitness_mutated = fitness_evaluation(mutated_population)

    to_sort = []
    for k in range(len(mutated_population)):
        to_sort.append([mutated_population[k], fitness_mutated[k]])
    mutated_sorted = sorted(to_sort, key = lambda x:x[1], reverse=True)

    input = []
    for k in range(20):
        input.append(mutated_sorted[k][0].copy())
    print(input[0])

print("Final Result:",input[0])
```

2.4 运行结果及分析

按照上面展示的代码，我们对产生的种群迭代了 100 次，得到了如下的音乐片段：

$$result = [1, 1, 6, 5, 6, 5, 5, 15, 4, 4, 3, 4, 3, 1, 0, 14, 5, 7, 3, 4, 2, 3, 2, 15, 6, 5, 4, 4, 4, 4, 1, 15,$$

$$2, 1, 4, 5, 6, 7, 6, 15, 4, 3, 2, 3, 2, 3, 1, 15, 5, 5, 4, 5, 4, 2, 2, 15, 5, 5, 5, 4, 3, 4, 3, 15]$$

由于产生的种群和各种操作均具有随机性，因此每次产生的结果是不一样的。我们可以看到，基于随机产生的初始种群，利用最为简单的适应度函数，我们确实得到了一个与参考旋律（Reference Melody）近似的旋律片段。

此外，我们还可以调整迭代次数 n 的值，以及各种变异及旋律变换操作中操作的次数。以迭代次数 n 为例，我们将 n 设置为 500，程序会进行更多次数的迭代，也会产生更好的结果；代价就是程序会占用更多的时间和空间。（n = 500 的结果如下所示）

$$result = [1, 1, 5, 5, 6, 6, 4, 15, 4, 3, 3, 3, 2, 2, 1, 15, 5, 4, 4, 4, 3, 3, 2, 15, 5, 5, 4, 5, 3, 3, 2, 14,$$

$$1, 1, 5, 5, 6, 6, 5, 15, 4, 3, 2, 2, 2, 2, 2, 15, 5, 5, 4, 4, 2, 3, 2, 15, 6, 5, 4, 4, 3, 3, 2, 15]$$

3 对于遗传算法的进一步完善：输出接口与适应度函数

有了对文献中最基础的遗传算法的复现后，我们希望能让程序的功能更加完善和强大：可不可以直接将结果转化为可以聆听的音频，让我们得以直接感受到机器作曲生成的结果的优劣，而不是面对一长串数字，为如何将它们转化为声音而头痛；仅有 8 个小节的音乐片段是不是太短了，我们可以进一步延长产生的音乐片段；可不可以对于某种适应度函数，直接产生某种风格的音乐；如何来完善适应度函数，让其对于机器生成的音乐片段的“美”与我们人类的听觉与主观感受上的“美”更加吻合？这一系列问题都将在这一小节给予一个答复。

3.1 Pysynth 及格式转化接口介绍

3.1.1 PySynth 介绍

PySynth 是一个开源的 python 库，可用于将 ABC、MIDI 等格式的简谱转化为具有各种乐器音色的.wav 音频文件，不同音色需使用 PySynth 的不同变种版本，比如 B,E 版本接近钢琴音色，S 接近吉他等弦乐器音色。PySynth 可通过”git clone” 指令下载后按其 README 说明安装。详细的使用说明可以参考网页：https://mdoege.github.io/PySynth/

在完善的遗传算法程序中（以下简称“本项目”）主要使用 PySynth 的 make_wav 函数生成.wav 音频文件。该函数需要指定输出文件名 fn。本项目输入的简谱格式是 PySynth 接受的一种简单的二维容器的变量，形如

$$[['c\#5*', -4], ['b5', 2]...]$$

下文中将该格式称为 syn 格式。

该二维容器内的各个元素为一个音名-时值对。比如上述例子中的 ['c#5*', 4],'c#5*' 第一个'c' 即音名 C, 'r' 为休止符；'#' 即升号，'5' 代表钢琴顺序的第五个八度，'*' 代表加重。时值-4 表示带附点的 4 分音符，负号表示附点。

3.1.2 简谱的数学表示的转化接口函数

下文中函数名均代表本项目中的定义的函数。

本项目中主要使用的简谱格式，0 15 组成的一维列表，0 代表休止符，1 14 为 C 大调第 4 及第 5 个八度内的音名，默认为 8 分音符，15 代表延长 1 个 8 分音符。下文中称该格式为 notation 格式。

一、最终音频文件输入格式转化接口

```
note2abc = ['r', 'c', 'd', 'e', 'f', 'g', 'a', 'b', 'c5', 'd5', 'e5', 'f5', 'g5', 'a5', 'b5']
temp = [8,4,-4,2,2,-2,-2,1] #8分音符为基准。
def note2syn(notation):#简谱notation转化为音名谱syn.不考虑升降号及重音，且不带附点)
    syn = []
    for note in notation:
        if note != 15:
            syn.append([note2abc[note],temp[0]])#默认0个15
        else:
            num15 = temp.index(syn[-1][1])+1
            syn[-1][1] = temp[num15]
    return syn
```

该函数逐个遍历 notation 中的数字，通过索引列表 note2abc 转化为对应音名。若遇到 15 则通过占 8 分音符时值的个数索引列表 temp 改变时值表示。

二、算法内部使用的格式转化接口

以上转化接口会使得原本的简谱丢失时值信息，具体来说，含 5 个或 7 个 8 分音符的时值无法通过单个附点表示。所以在以上接口仅在最终转化.wav 音频时使用，计算过程中使用的是以下函数。

```
def note2syn_time(notation):
#简谱notation转化为音名谱syn.不考虑升降号及重音，时值直接由含15的个数+1表示）
    syn = []
    for note in notation:
        if note != 15:
            syn.append([note2abc[note],1])
        else:
            syn[-1][1] += 1
    return syn
```

该函数输出的音名-时值对中的时值不再是几分音符的表示，而是含 8 分音符时值的个数。该格式下文称为 syn_time。转化回简谱序列使用如下函数：

```
#syn转化回简谱
def syn_time2note(syn_time):
    notation = []
    for note,time in syn_time:
        notation.append(note2abc.index(note))
        notation += [15]*(time-1)
    return notation
```

该格式在遗传算法中的一些操作中非常实用，因为很多操作仅处理有效音名，比如遗传的交叉操作，计算音程关系。避免了寻找 15 前的音名的复杂操作。

3.2 适应度函数

```
def fitness_evaluation(mutated_population):
    fitness_value = []
    for item in mutated_population:
        #小节为单位分析
        scores = np.array([structure_evl(item),whole_evl(item)])
        fitness_weight = np.array([0.9,0.1])
        fitness = (fitness_weight*scores).sum()
        fitness_value.append(fitness)
    return fitness_value
```

本项目使用的适应度函数分为两个主要部分：整体性评分和结构性评分。各个部分又有很多分量。假设各个分量评分记为 s_{ij}，各个分量权重为 λ_{ij}，则最后总的适应度函数值为 $\Sigma\lambda_{ij}s_{ij}$. 其中 i 为整体或结构指标，j 为子级分量的指标。

3.2.1 整体评分

整体评分包括参考旋律评分、重复度评分、自和谐度评分。注意重复度评分的权重是负值。

```
def whole_evl(notation):
    syn = note2syn_time(notation)
```

```
    inner_harm_score = inner_harm(syn)
    repeat_score = repeat_evl(syn)
    refer_score = refer_evl(notation)

    scores = np.array([repeat_score, inner_harm_score, refer_score])
    whole_weight = np.array([-0.5,1.2,0])
    whole_score = (scores*whole_weight).sum()
    return whole_score
```

一、参考旋律评分

参考旋律评分主要依据某个体片段相对参考旋律的欧式距离的倒数，并作归一化处理。

```
def refer_evl(notation):
    #reference
    new_melody = np.array(notation)
    max_norm = 8*2
    error = np.linalg.norm(new_melody-ref_melody)
    refer_score = 1/(error+1)*max_norm
    return refer_score
```

二、重复度评分

```
#重复度的估计
def repeat_evl(notation):
    score = 0
    repeat_num = 1
    for index in range(1,len(notation)):
        note_i = notation[index]
        note_pre = notation[index-1]
        if note_i == note_pre:
            repeat_num += 1
            score += int(repeat_num>2)*3 + int(repeat_num>4)*1000 #限制三个重复；超过5个直接截断（尤
                                                                  其是15，0）
            print(repeat_num)
        else:
            repeat_num = 1
    return score/len(notation)
```

该函数输入 notation 格式，遍历以统计连续的字符。作者认为 3 个及以上的重复音符会使音乐单调无味，所以当重复度达到 3 及以上，每增加重复度一次就记三分。针对休止符 0 和延长符 15，为了保证算法稳定性（见之后的结构性评分），通过记 1000 分直接截断重复超过 4 个的情况。最后将分数用简谱总长度归一化后输出。

三、自和谐度评分

首先定义谐和度评分表，即对不同音程关系给予不同评分。

```
#和谐度评分表（按音程差）：
pitch_harm = {
    1:5,
    8:5,#纯一度、纯八度
    4:4,
    5:4,#纯四、五
    3:3,#大小三
    6:2,#大小六
    2:1,
    7:1#大小二、七
```

```
}
```

然后根据单音间的音程差计算和谐度。

```
#两个单音间的和谐度
def note_harm(note1,note2):
    pitch_diff = ( note2abc.index(note1) - note2abc.index(note2) )%7 + 1
    #注意'c5','d'计算结果会是7，侧面反映这种算法互补的音程关系某种意义上等价；
    #或说如果设置互补音程谐和度不同，则八度音符不等价
    return pitch_harm[pitch_diff]
```

自和谐度函数遍历一个音乐片段，不考虑休止符，计算每个音与前一个音的单音谐和度，然后根据时值加权平均。最后归一化输出。

```
#一个单音列的内部和谐度
def inner_harm(individual):#输入的individual是pysynth中的二元表示法；
    score = 0
    max_score = 0
    min_score = 0
    for index in range(1,len(individual)):
        note_i, time = individual[index]
        note_pre, time_pre = individual[index-1]
        if note_i[0] != 'r': #暂不考虑休止符
            score += note_harm(note_pre,note_i)*time*time_pre
            max_score += 5*time*time_pre
            min_score += time*time_pre
    return (score-min_score)/max_score #归一化
```

3.2.2 结构性评分

结构性评分考虑小节内及小节间的关系，需要将简谱按小节切片，直接丢弃开头的 15 延长符。得到的评分分量可能是关于小节的，也可能是小节间的。小节间的评分要回到该函数进行综合计算。具体来说有节奏旋律的评分和和弦评分。

```
def structure_evl(notation):
    bar_num = len(notation)//8
    chord_levels = []
    max_chord_evls = []
    mel_rhy_score = 0
    for bar_index in range(bar_num):
        bar = list(notation[bar_index*8:(bar_index+1)*8])
        while bar[0] == 15:
            bar.pop(0)
        syn = note2syn_time(bar)
        chord_level, max_chord_evl = chord_cognize(syn)
        chord_levels.append(chord_level)

        mel_rhy_score += melody_rhythm_match(syn)
        max_chord_evls.append(max_chord_evl)

    chord_score = chord_evl(chord_levels,max_chord_evls)

    scores = np.array([chord_score,mel_rhy_score])
    structure_weight = np.array([0.5,0.5])
    structure_score = (scores*structure_weight).sum()
```

```
    return structure_score
```

一、节奏旋律评分

```
#小节内的旋律、节奏按八分为主的升调式
ref_melody_method = [1,1,1,0,1,1,0] #是否升调
ref_rhythm = [1,1,1,1,1,1,2]
mel_rhy_log = []
def melody_rhythm_match(syn):
    melody_dis = 0
    rhythm_dis = 0
    for i in range(1,len(syn)):
        melody_dis += (int((note2abc.index(syn[i][0])-note2abc.index(syn[i-1][0]))>0) -
                                                    ref_melody_method[i-1])**2
        rhythm_dis += (ref_rhythm[i-1]-syn[i][1])**2
    mel_rhy_scores = np.array([1/(melody_dis+1) , 1/(rhythm_dis+1)])*7 #max_norm=7
    mel_rhy_weight = np.array([0.5,0.5])
    mel_rhy_score = (mel_rhy_scores*mel_rhy_weight).sum()

    mel_rhy_log.append(mel_rhy_scores)
    return mel_rhy_score
```

我们认为小节内反复升降音高的音乐不好听，所以定义了一个升调为主的旋律形式 ref_melody_method = [1,1,1,0,1,1,0] 作为参照，1 表示升调，0 表示降调。即每半小节连续升调之后降调。

我们认为节奏需要长短结合，这里采用的是先短后长的节奏型作为参照 ref_rhythm = [1,1,1,1,1,1,2] 两种评分分量即某小节的旋律、节奏型与参考的形式的欧式距离的倒数。

二、**和弦评分** 和弦评分的关键在于 structure_evl 中对各小节进行和弦识别。

首先定义“和弦和谐度”：即遍历一个和弦包含的音即某个体包含的音，对时间加权平均归一化即和弦和谐度。

```
#一个单音列与和弦的平均谐和度
def chord_harm(chord,individual):#输入的individual是pysynth中的二元表示法;chord仅由音名组成
    score = 0
    max_score = 0
    min_score = 0
    for note_c in chord:
        for note_i,temp in individual:
            if note_i[0] != 'r': #暂不考虑休止符
                time = temp
                #dot = 1 if temp > 0 else -3/2
                #time = dot/temp #pysynth, 时值作为权重,temp中的数字比如4, 代表4分音符; 负号表示附
                                                                    点, 使得时长变为原来的3/2
                score += note_harm(note_c,note_i)*time
                max_score += 5*time
                min_score += time
    if max_score == 0:
        max_score = 1
    return (score-min_score)/max_score #归一化
```

然后定义“和弦识别”：通过计算该小节对 C 大调 7 级和弦的和弦和谐度，记录和谐度最大值及对应的和弦级数，认为该和弦级数即该小节的和弦（比如伴奏和弦）。

```
#和弦识别
chord_sum = 7
chords = [np.array(c2b)[[i%7,(i+2)%7,(i+4)%7]] for i in range(chord_sum)]#七级三和弦
```

```
def chord_cognize(syn):
    #和弦估计
    chord_evl = [chord_harm(chord,syn) for chord in chords]
    max_chord_evl = max(chord_evl)
    chord_level = chord_evl.index(max_chord_evl)
    return chord_level, max_chord_evl
```

我们还可以在此基础上，测试一下和弦与和弦之间的“和弦和谐度”：

```
#和弦识别实验
chord_level = 7
chords = [np.array(c2b)[[i%7,(i+2)%7,(i+4)%7]] for i in range(chord_level)]
chord_evls = []
for index in range(chord_level):
    indiv = [[note,1] for note in chords[index]]
    chord_evl = [chord_harm(chord,indiv) for chord in chords]
    chord_evls.append(chord_evl)
    chord_index = chord_evl.index(max(chord_evl))
    print(chord_index==index)
以上实验测试和弦与和弦之间的和弦和谐度
from seaborn import heatmap
heatmap(np.array(chord_evls),xticklabels=c2b,yticklabels=c2b)
```

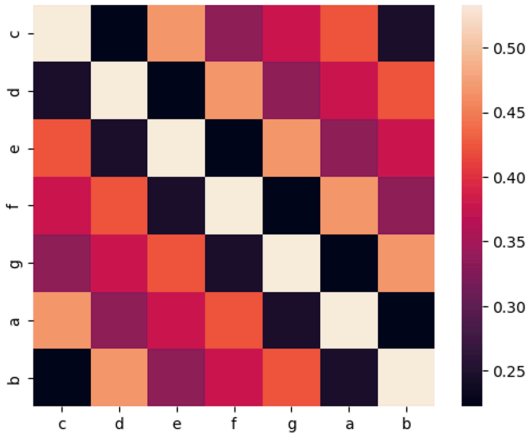

图 1: 实验结果

可以看到，和弦和谐度表现出规律性，首先和弦自身和谐度最高，约 0.5，相邻的和弦和谐度最低，约 0.25. 这证实了和弦和谐度可以用于和弦识别。

最后，我们可以讨论“和弦评分”了：和弦评分分量为和弦演进评分和和弦匹配评分。和弦演进评分即计算该个体各个小节的和弦与参考和弦的欧式距离的倒数，和谐匹配评分即对各小节对其和弦的和弦和谐度的平方平均，作者认为和弦和谐度越高，该小节的和弦约清晰分明。

```
refer_chord_evolution = ['c','f','e','a','f','g','c','c']
refer_chord_evolution_level = np.array([c2b.index(refer) for refer in refer_chord_evolution])
#和弦相关表现的估计
def chord_evl(chord_levels,max_chord_evls):
    #chord_evolution = inner_harm([[c2b[chord_level],1] for chord_level in chord_levels]) #演进要求
                                                                                          和谐
    chord_ev_dis = np.linalg.norm(np.array(chord_levels)-refer_chord_evolution_level)
    chord_evolution = 1/(1+chord_ev_dis)*8 #norm
    chord_match = np.linalg.norm(max_chord_evls)*2
```

```
    chord_scores = np.array([chord_evolution,chord_match])
    chord_weight = np.array([0.4,0.5])
    chord_score = (chord_scores*chord_weight).sum()

    chord_log.append(chord_scores)
    return  chord_score #每小节
```

4 对完善的遗传算法程序的运行结果分析

4.1 实验设置

输入的种子是董小姐旋律和小星星旋律各一半的组合。注意为了保持稳定性，董小姐片段末尾做了重复倒数第二小节的片段的处理。

```
#董小姐旋律
dong = [0, 15, 15, 15, 5, 15, 8, 6, 15, 15, 15, 15, 0, 6, 6, 6, 7, 15, 5, 15, 5, 15, 6, 5, 15, 3, 15
                                                    , 15,
0, 2, 3, 2, 3, 15, 2, 5, 5, 15, 15, 15, 0, 15, 15, 2, 5, 15, 15, 2, 5, 3, 15, 15, 5, 3, 15, 15, 5, 3
                                                    , 15, 15, 5, 3, 15, 15]
# ps.make_wav(note2syn(dong),fn="dong.wav")

reference_melody = [1,1,5,5,6,6,5,15,4,4,3,3,2,2,1,15,5,5,4,4,3,3,2,15,5,5,4,4,3,3,2,15,
1,1,5,5,6,6,5,15,4,4,3,3,2,2,1,15,5,5,4,4,3,3,2,15,5,5,4,4,3,3,2,15]

input = [dong.copy() for _ in range(5)] + [reference_melody.copy() for _ in range(5)]
result = genetic(input,100)
ps.make_wav(note2syn(result),fn="dong_test.wav")
```

4.2 可视化适应度函数分量

接下来将各适应度函数的分量可视化。在实验过程中记录了每次调用相应分量评分函数时的分量值，也就是每次迭代对种群每个个体进行计算适应度函数就记录一次。展示的是平均后每次迭代种群的适应度函数评分。

4.2.1 节奏旋律评分

可以看到基于小节内旋律节奏型定义的评分一开始便迅速提升，在迭代次数达 20 之后保持在 1 左右。之所以如此稳定是因为它关于不同小节是独立的，而且相对于其他评分分量是独立的。比如对于旋律而言，仅是小节内交换音符位置不影响自和谐度，也不影响小节和弦匹配度和和弦演化。

4.2.2 和弦评分

可以看到和弦匹配度有一定提升。主要是和弦演进评分阶段性提高，比如大概 85 次迭代出突然上升。个人认为由于和弦演进是小节间的关系，所以主要是遗传交叉操作能改变该分量，能获得比较大的突然提升。如果仅靠单音变异和交换，很难一次优化和弦演进。

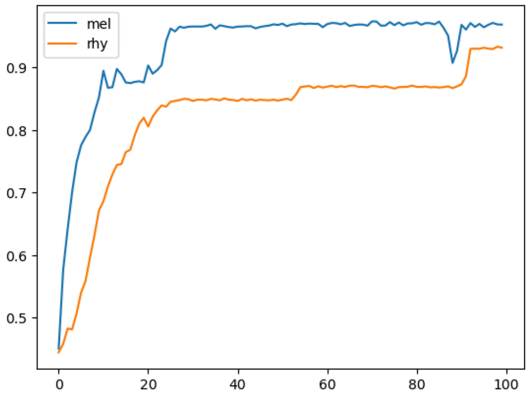

图 2: 节奏旋律评分

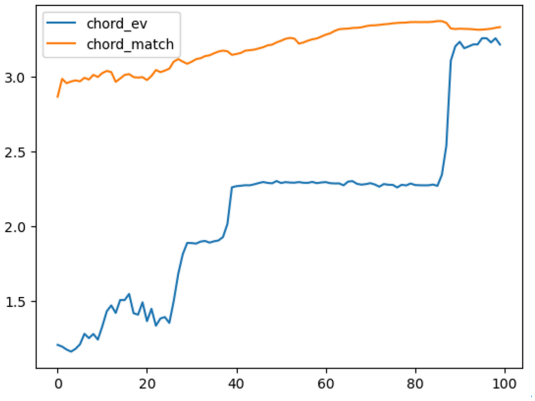

图 3: 和弦评分

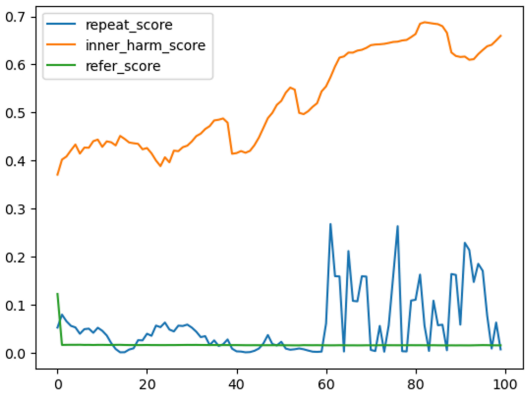

图 4: 整体性评分

4.2.3 整体性评分

可以看到，由于一开始的种子含一般参考旋律小星星，所以参考分比较高，之后就远离了参考旋律。自和谐度在 50 次迭代之后迅速上升，提升接近一倍。重复度分数在一开始被压低，在 60 之后开始反复升降。

4.3 稳定性分析

算法稳定性指的是算法抵抗干扰而不偏离最优解或向最优解演化的过程。演化过程由上一节可视化，可以看到大部分分量都是稳定的增大或减小，朝着适应度函数要求的方向进化，说明算法的稳定性较好。

另外，算法本身设计有一些的数据结构上的不稳定性，比如说在结构性评分中，需要逐小节处理。而小节的定义是 8 个一组的 notation 去掉开头的 15。如果一个小节全是 15，那么算法会报错。这也是为什么董小姐旋律片段需要处理，否则若一个小节仅由一个音组成，当这个音变异为 15 时程序将报错。重复度评分去重能一定程度缓解该问题，但不能杜绝。

4.4 音频结果示例

第一次迭代生成的音频：

见附件 3

不难听出该片段几乎是董小姐和小星星拼接而成。主要是遗传中的交叉操作起作用。

第 100 次迭代生成的结果：

见附件 4

已经听不出董小姐和小星星的旋律。整体来说比较和谐，而且可以听出来各小节基本符合参考的旋律和节奏型，先升调几次再降调，以及先短音再一次长音。

总的来说，实验结果比较理想，达到了适应度函数的设计初衷。之后可以进一步优化适应度函数，比如旋律和节奏和和弦演进更灵活的定义等。

5 遗传算法的代码

5.1 基础的遗传算法代码

源代码（by 郝炜天 and 江泽辉）：见附件 1

5.2 优化的遗传算法代码

源代码（by 陈鹏）：见附件 2

6 总结和展望

在本次研究中，我们采取了“三步走”的方式来完成本次的研究课题：遗传算法的建立与完善。首先，我们仔细阅读了作业文档中提供的参考文献 [1]，并根据此参考文献检索了更多同类型的参考文献；总结了这些文献中的共同点，并对参考文献 [1] 中的遗传算法的优点和缺点提出了继承和改进的想法。其次，我们利用 python 编程建立了基本的遗传算法程序，并得到了较为满意的结果。最后，在基本的遗传算法程序的基础上，我们增加了音频的输出接口，以及对于适应度函数评判准则的进一步细化和改良，实现了很好的机器作曲结果。

虽然结果喜人，但我们的研究仍存在一些没有覆盖到的点：比如我们只能处理某种调式的音乐片段，生成的也是该调式的音乐片段；对于乐段特定节奏的讨论没有深入地进行等等。我们也对此提出了可能的解决方案：我们可以在初始种群的设定中加入某种模版——譬如调性的模版和节奏的模版。规定好了模版之后，我们可以在诸多模版中选择自己所喜好的那种，让机器进行作曲。

总的来说，我们的研究是比较成功的，但也仍然有很大的进一步发展的空间。

7 参考文献

[1] Dragan Matic, A genetic algorithm for composing music, Yugoslav Journal of Operations Research, 20 (2010), 157–177.

[2] Biles, J.A., "Improvizing with genetic algorithms: GenJam" , Evolutionary Computer Music (Eduardo Reck Miranda and John Al Biles (Eds)), Springer, 2007.

[3] Towsey, M., Brown, A., Wright, S., and Diederich, J., "Towards Melodic Extension Using Genetic Algorithms" , Educational Technology & Society, 4 (2) 2001.

[4] Gartland-Jones, A., and Copley, P., "The Suitability of Genetic Algorithms for Musical Composition" , Contemporary Music Review, 22 (3) (2003) 43–55.

[5] Gartland-Jones, A., and Copley, P., "The Suitability of Genetic Algorithms for Musical Composition" , Contemporary Music Review, 22 (3) (2003) 43–55.

[6] 王鑫, 王永滨, 吕志胜, 李樱, 吴林. 一种基于混合算法的江南小调计算机辅助作曲的方法, 国家发明专利,2019.

[7] 叶煦舟. 基于遗传算法的和弦伴奏生成方法，国家发明专利，2015.

[8] Ralley, D., "Genetic algorithm as a tool for melodic development" , Proceedings of the 1995 International Computer Music Conference, ICMA, San Francisco, 1995.

[9] Towsey, M., Brown, A., Wright, S., and Diederich, J., "Towards Melodic Extension Using Genetic Algorithms" , Educational Technology & Society, 4 (2) 2001.

[10] Wiggins, G., Papadopoulos, G., Phon-amnuaisuk, S., and Tuson, A., "Evolutionary Methods for Musical Composition" , Proc. of the CASYS98 Workshop on Anticipation, Music&Cognition, 1998.