Poisson.h 文档说明

Haochen Huang

版本：4.02test

日期：2023 年 8 月 3 日

目录

1 理论背景 2

1.1 Poisson 方程由来 . . . 2

1.2 Relaxation 函数由来 . . . 2

1.3 Jacobi 迭代 . . . 2

2 具体算法 3

3 源代码解析 3

3.1 源代码构成及结构简介 . . . 3

3.2 Multigrid cycle 函数 . . . 4

3.3 Multigrid solver 函数 . . . 6

3.4 重要计算函数的构建 . . . 9

3.5 具体使用示例 . . . 13

4 附录：Basilisk 网格生成与插值 14

4.1 leaf cell . . . 14

4.2 网格排布规律 . . . 14

4.3 网格点插值与平均 . . . 16

参考文献 17

摘　要

本文为 basilisk 的头文件 Poisson.h 的说明文档。

2.02 更新：更新模板添加高亮、索引，更新文章内容，修正某些显示错误并添加程序结构图及说明。

3.02 更新：更正 $relax$ 函数中 $Jacobi$ 与 GS 迭代之间的关系，回顾之前相关使用函数指针进行输入的地方，指出改装求解器的可能条件。

4.02 更新：添加不同层级网格迭代的相应细节，添加 Basilisk 网格的判定方式及相应数据点的插值方法。

1. 理论背景

相关细节参考 Basilisk 原作者著作 [2][1]

1.1 Poisson 方程由来

我们可以由 bcg.h 中的对流方程求出一个速度场 U^{**}，由于速度物理特性，其必然是光滑且连续的，由 Hodge 分解可得，该速度场可以分为一个无散场与无旋场即：

$$U^{**} = U + \nabla\phi \tag{1}$$

其中 $\nabla U = 0$ 这样 U 就既满足动量方程又满足不可压方程，为真实速度解。

1.2 Relaxation 函数由来

求解 Poisson 方程

$$\nabla^2\phi = \nabla U^{**} \tag{2}$$

针对某一个单元，在单元内部对上式进行体积分有：

$$\int_{\partial\mathscr{B}} \nabla\phi \cdot \mathbf{n} = \int_{\mathscr{B}} \nabla U^{**} \tag{3}$$

在单元上进行离散有：

$$\sum_d h\nabla_d\phi = h^2\nabla \cdot U^{**} \tag{4}$$

其中下角标 d 表示方向，在离散过程中等式左边数据储存于单元面中心，而等式右边则是单元体中心。在 bcg.h 中我们已经利用已知的 u^n 将 U^{**} 计算出来了，现在的目标则是计算 ϕ。

为了得到 ϕ，我们假设 ϕ 在单元表面上的梯度值 $\nabla_d\phi$ 可以用在单元中心的 ϕ 线性表示，即：

$$\nabla_d\phi = \alpha_d\phi + \beta_d \tag{5}$$

其中 α, β 均为常数，其值根据周边网格情况等可以计算出来，暂时按下不表。由此我们可以得到相应 ϕ 值表达式

$$\sum_d h\nabla_d\phi = \sum_d \alpha_d\phi + \sum_d \beta_d \tag{6}$$

即有：

$$\mathscr{R}(\phi, \nabla U^{**}) = \phi \leftarrow \frac{h\nabla \cdot U^{**} - \sum_d \beta_d}{\sum_d \alpha_d} \tag{7}$$

即能够构成 $\mathscr{L}(A) = B$ 的 A, B 都可以满足 Relaxation 函数关系。

1.3 Jacobi 迭代

在构建完 Relaxation 函数后，我们将对其使用 Jacobi 迭代，以完成相应求解。

假设我们有方程组

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{11}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \quad \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{8}$$

不考虑收敛问题，我们将 $\mathbf{x}^0 = x_1^0, x_2^0 \cdots x_n^0$ 依次带入方程求解得到 $\mathbf{x}^1 = x_1^1, x_2^1 \cdots x_n^1$，依次迭代，依据相应理论，在迭代次数足够多的情况下，解最终会收敛于方程真解。

根据之前构建的 Relaxation 函数，由于 ∇U^{**} 已知，我们可以对场量 ϕ 采取类似的迭代方式，最终得到目的解。

2. 具体算法

$\mathscr{L}$ 为线性算子即：

$$\mathscr{L}(\phi + \delta\phi) = \mathscr{L}(\phi) + \delta\phi \tag{9}$$

其中 $\mathscr{L}(\phi + \delta\phi) = \nabla U^{**}$ 而 ϕ 为预测值，那么 $\delta\phi$ 就是预测值与真实值之差，令残差 R 为

$$R = \nabla U^{**} - \mathscr{L}(\phi) \tag{10}$$

由此 $\delta\phi$ 与 R 构成了拉普拉斯算对，他们之间的关系就可以用 Relaxation 函数 $\mathscr{R}$ 来表示。

接下来介绍 Basilisk 的自适应网格模式。

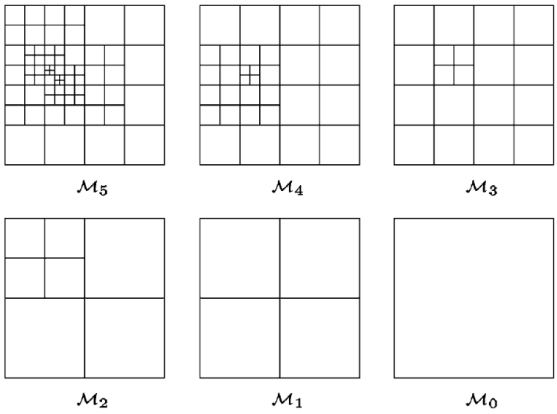


图 1: 二维情况下，同一单元网格级别变化情况，摘自 [2]

如图1中所示，其中 Ml 表示网格加密级别；注意，这是同一个网格的不同加密形式。l 层的残差可以由 $l + 11$ 计算得出即

$$R_l = \frac{\sum_i h^2 R_{l+1}}{\sum_i h^2} \tag{11}$$

3. 源代码解析

3.1 源代码构成及结构简介

Poisson.h 主要求解的方程为广义的 Poisson–Helmholtz 即：

$$L(a) = \nabla \cdot (\alpha \nabla a) + \lambda a = b \tag{12}$$

源代码一共分为四个部分，分别是：

1. Multigrid cycle 函数的构建3.2，此函数表示算法中的第二个 for 循环，即在已知各个级别网格的残差后，通过 $\mathscr{R}$ 函数进行迭代，并向高一层级网格进行差分，得到该网格 δa 的初始值，该函数并没有 $\mathscr{R}$ 的定义

Algorithm 1: Poisson solver 框架

Input: 初始 ϕ^0，最密网格 M_L

Output: 相应计算区域中全场 ϕ 值

由 ϕ^0 计算再 M_L 上的残差 R_L

while $|\alpha R_L|_\infty < \epsilon$ **do**

  for $l = L-1; l \geq 0; l--$ **do**

    由 R_{l+1} 计算 R_l;

  在最低级网格 M_0 上使用 $\mathscr{R}(\delta\phi, R_0)$ 直到最后的值收敛;

  for $l = 1; l \leq L; l++$ **do**

    将在 M_{l-1} 上获得的 $\delta\phi$ 在空间上进行插值后直接作为初始猜测值进行 $\mathscr{R}(\delta\phi, R_l)$ 的 n 次迭代;

  将迭代计算出的 $\delta\phi$ 附加在原本 M_L 的 ϕ 值上;

  再次计算 R_L;

return ϕ;

2. Multigrid solver 函数构建3.3，该函数与 Multigrid cycle 函数一同构成完整的算法循环框架（Multigrid cycle 将会在其中被引用），该函数负责判断计算域内的最大残差是否满足标准，以及当某些限定情况发生时，对迭代次数进行改变
3. 两个重要功能函数的构建3.4：$relax$ 函数即 $\mathscr{R}$，以及计算残差的 residual 函数，这两个函数在之前构建的整体框架中均被引用。
4. 整体合并解决问题的示例3.5

其相互关系如下图所示：由图所示，mg solver 为求解器的主体部分，其主要功能在于调配各个函数的调用顺序，判断是否满足输出标准，以及调整循环的次数，而 $mg\,cycle$ 与 $relax$ 函数则构成了一次或数次（基于 $mg\,solver$ 传入参数调控）完整的 G-S 迭代，并更新初始值，并将结果传递至 $residual$ 计算残差，$residual$ 返回残差供 $mg\,solver$ 进行判断，选择调整参数再次进入循环或者跳出循环输出结果。

3.2 Multigrid cycle 函数

函数输入计算目标指针 a，残差指针 res，修正目标值指针 da，以及 $\mathscr{R}$ 函数指针 $relax$ 等进行从最低层级向最高层级网格的残差修正。

注意此处输入值中包含函数指针，修改者可以通过传入具有相同参变量的函数达到改装求解器的目的。

```
void mg_cycle (scalar * a, scalar * res, scalar * da,
               void (* relax) (scalar * da, scalar * res,
                    int depth, void * data),
               void * data,
               int nrelax, int minlevel, int maxlevel)
{
```

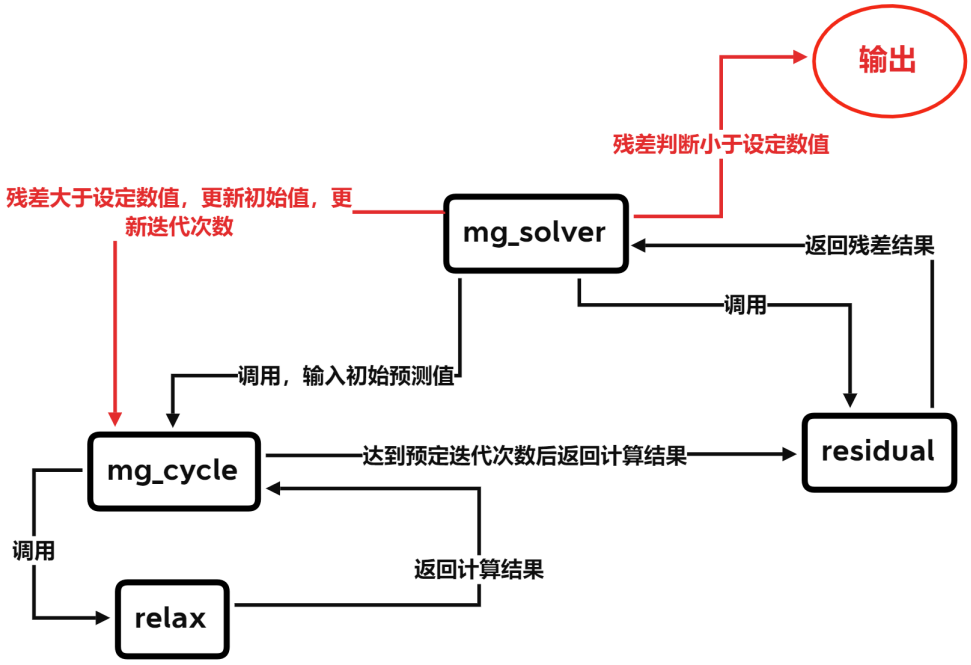


图 2: poisson.h 头文件中主要函数关系

```
7  restriction (res);//利用平均计算每一层级网格上残差值
8
9  //从最低的网格密度层级开始对相应的计算区域进行 Jacobi 迭代
10 minlevel = min (minlevel, maxlevel);
11 for (int l = minlevel; l <= maxlevel; l++) {
12
13   if (l == minlevel)
14     foreach_level_or_leaf (l)//对当前级别网格，或小于 l 级但已经是 leaf 的网
       ↪ 格进行遍历
15       for (scalar s in da)
16           foreach_blockf (s)//默认在 common.h 中定义为空，具体定义在
             ↪ grid/layers.h 中，用法应该与存储数据方式相关，用于快速遍历内存上
             ↪ 值，测验后发现其存在对结果并没有影响
17               s[] = 0.;
18   else
19     foreach_level (l)
20       for (scalar s in da)
21           foreach_blockf (s)
22               s[] = bilinear (point, s);
23   //在轮到 l 级网格时，由于网格加密导致网格中心位置变动，需要对上一级中求出的
     ↪ da 进行插值，并将其作为本轮 Jacobi 迭代的初始值
```

```
    boundary_level (da, l);
    for (int i = 0; i < nrelax; i++) {
      relax (da, res, l, data); //利用 ℛ 函数进行 Jacobi 迭代，迭代次数为输入值 nrelax，该数值在后面会视情况而改变。
      boundary_level (da, l);
    }
  }

  foreach() {
    scalar s, ds;
    for (s, ds in a, da)
      foreach_blockf (s)
        s[] += ds[];//更新最密网格中的目标数值 a
  }
}
```

由此，算法中的 Jacobi 迭代进行完毕。

3.3 Multigrid solver 函数

该函数通过在其中通过引用 $Multigrid\,cycle$ 构建完整的算法结构，其首先定义两种数据结构，首先是表示整个计算域状态的数据结构 $mgstats$：

```
int NITERMAX = 100, NITERMIN = 1;
double TOLERANCE = 1e-3;//默认定义，包括迭代次数与残差宽容度

typedef struct {
  int i;              // 总循环次数
  double resb, resa;  // 在迭代前后残差的最大值，用于结束循环判断
  double sum;         // sum of r.h.s. (r.h.s. 代表等式右端，即所有的 b 的和)
  int nrelax;         // Jacobi 循环 ℛ 次数
  int minlevel;       // 网格密度的最低密度等级
} mgstats;
```

以及所需要求解目标函数的数据结构：

```
struct MGSolve {
  scalar * a, * b;
  double (* residual) (scalar * a, scalar * b, scalar * res,
```

```
              void * data);// 求解残差的函数，在下一节会出现
  void (* relax) (scalar * da, scalar * res, int depth,
          void * data);
  void * data;// 该参数的具体细节将在下一节展示

  int nrelax;
  scalar * res;
  int minlevel;
  double tolerance;
};
```

我们构成的函数 mg_solve 返回值的数据类型为 mgstats 即循环状态，输入值则为 MGSlove 型数据 p

```
mgstats mg_solve (struct MGSolve p)
{
  //根据输入数据格式的 a 分配修正数据的指针，即保证 da 与其有相同的分布与边界条件
  scalar * da = list_clone (p.a), * res = p.res;
  if (!res)
    res = list_clone (p.b);

  /**
  The boundary conditions for the correction fields are the
  *homogeneous* equivalent of the boundary conditions applied to
  *a*. */

  for (int b = 0; b < nboundary; b++)
    for (scalar s in da)
      s.boundary[b] = s.boundary_homogeneous[b];

  // s 即为要返回的 mgstats 型数据，先对其进行初始化
  mgstats s = {0};
  double sum = 0.;
  foreach (reduction(+:sum))
    for (scalar s in p.b)
      sum += s[];
  s.sum = sum;//对等式右端 (r.h.s.) 进行赋值
  s.nrelax = p.nrelax > 0 ? p.nrelax : 4;//默认的 ℛ 迭代次数为 4，如果自带则使
  ↪ 用自带参数
```

```
25
26    //使用 residual 函数计算该计算域内最大残差并赋值
27    double resb;
28    resb = s.resb = s.resa = p.residual (p.a, p.b, res, p.data);
29
30    //设定残差判定的默认值
31    if (p.tolerance == 0.)
32      p.tolerance = TOLERANCE;
33
34    //开始进入算法中的 while 循环
35    for (s.i = 0;
36         s.i < NITERMAX && (s.i < NITERMIN || s.resa > p.tolerance);
37         s.i++) {
38      //引用 mg cycle 函数，开始进入 Jacobi 迭代循环
39      mg_cycle (p.a, res, da, p.relax, p.data,
40            s.nrelax,
41            p.minlevel,
42            grid->maxdepth);
43
44      //再次计算残差，用于判断
45      s.resa = p.residual (p.a, p.b, res, p.data);
46
47  //进入对 Jacobi 迭代次数的调整
48  #if 1
49      if (s.resa > p.tolerance) {
50        if (resb/s.resa < 1.2 && s.nrelax < 100)//如果在经历迭代后的残差 R_a 与迭
          ↪ 代前的残差 R_b 之比小于 1.2，则扩大迭代次数
51      s.nrelax++;
52        else if (resb/s.resa > 10 && s.nrelax > 2)//如果残差收敛效果好，则适当减
          ↪ 少迭代次数，加速计算
53      s.nrelax--;
54      }
55  #else
56      if (s.resa == resb) /* convergence has stopped!! */
57        break;
58      if (s.resa > resb/1.1 && p.minlevel < grid->maxdepth)
59        p.minlevel++;//如果迭代效果不理想，且网格密度大于迭代过程中的最小网格级别，
          ↪ 则抬高该级别
```

```
#endif

    resb = s.resa;
  }//while 循环终止处
  s.minlevel = p.minlevel;

  /**
  If we have not satisfied the tolerance, we warn the user. */

  if (s.resa > p.tolerance) {
    scalar v = p.a[0];
    fprintf (ferr,
         "WARNING: convergence for %s not reached after %d iterations\n"
         "  res: %g sum: %g nrelax: %d\n", v.name,
         s.i, s.resa, s.sum, s.nrelax), fflush (ferr);
  }//在数次循环后依旧无法满足判定标准，而是达到最大循环次数才实现跳脱循环，则需要向标准输出端发出警告。

  /**
  We deallocate the residual and correction fields and free the lists. */
  //清除栈
  if (!p.res)
    delete (res), free (res);
  delete (da), free (da);

  return s;
}
```

3.4 重要计算函数的构建

我们在之前两节中完整的构建了算法循环，本节我们来解析其中最重要的两个函数，$relax$ 与 $residual$。首先讲解 $relax$ 函数。

$relax$ 函数的目的是根据输入的数据对位于 (i,j) 上的 a 进行计算，对12进行离散有：

$$\frac{\alpha_{i+\frac{1}{2},j}\frac{a_{i+1,j}-a_{i,j}}{\Delta}-\alpha_{i-\frac{1}{2},j}\frac{a_{i,j}-a_{i-1,j}}{\Delta}}{\Delta}+\frac{\alpha_{i,j+\frac{1}{2}}\frac{a_{i,j+1}-a_{i,j}}{\Delta}-\alpha_{i,j-\frac{1}{2}}\frac{a_{i,j}-a_{i,j-1}}{\Delta}}{\Delta}+\lambda a_{i,j}-\gamma=b \tag{13}$$

$$a_{i,j}^{n+1}=\frac{\alpha_{i+\frac{1}{2},j}a_{i+1,j}+\alpha_{i-\frac{1}{2},j}a_{i-1,j}+\alpha_{i+\frac{1}{2},j+\frac{1}{2}}a_{i,j+1}+\alpha_{i,j-\frac{1}{2}}a_{i,j-1}-b\Delta^2-\gamma\Delta^2}{\alpha_{i+\frac{1}{2},j}+\alpha_{i-\frac{1}{2},j}+\alpha_{i,j+\frac{1}{2}}+\alpha_{i,j-\frac{1}{2}}-\lambda\Delta^2} \tag{14}$$

可以看到在 2 维情况下我们使用五点插值计算当地的 a 值，由此可以进行迭代。需要强调的是，

由于方程12的线性特性，将上式中的 a, b 置换成 da, R 方程依旧成立，而在绝大多数的情况下下文中的计算都针对后者。

上式针对形状大小相同的规则网格，在如图1所示网格中则需要进行插值或平均以满足相应网格的计算标准。(详见附录)

此处另一个值得注意的细节为迭代方式的选取，若无提前指定，函数迭代模式为 $Gauss-Seidel$ 迭代，反之则为 $Jacobi$ 形式的松弛因子迭代。具体实现方式为，设定中间变量 c，当没有提前声明时 c 会直接指向输入场 a，由此 (14) 中右侧的部分数值会随着遍历的顺序变为 $n+1$ 次循环值；而当进行 $Jacobi$ 时 c 会成为新定义的 $scalar$ 型数据，计算的结果会首先储存在其中，待计算完成后统一更新，从而保证了 (14) 左端为第 n 次的循环值。

```
struct Poisson {
  scalar a, b;
  (const) face vector alpha;
  (const) scalar lambda;
  double tolerance;
  int nrelax, minlevel;
  scalar * res;
#if EMBED
  double (* embed_flux) (Point, scalar, vector, double *);
#endif
};
```

此数据类型即在前文中出现过的 $data$ 数据类型，对方程12中的重要参数 λ, α 做了详尽定义

```
static void relax (scalar * al, scalar * bl, int l, void * data)
{
  scalar a = al[0], b = bl[0];//需要注意的是，relax 函数给了别的 a,b 的接口，而
  ↪ 并非直接从 data 型数据中提取，在正式的算法中，这两值分别为修正值 da 以及残
  ↪ 差 R
  struct Poisson * p = (struct Poisson *) data;
  (const) face vector alpha = p->alpha;
  (const) scalar lambda = p->lambda;
  //此为迭代模式的选择，当选择 JACOBI 后，对 a 的迭代会变为 1/3 a + 2/3 c，而如果并不选
  ↪ 择该模式，则会直接用新计算出来的数值取代
#if JACOBI
  scalar c[];
#else
  scalar c = a;
#endif
```

```
13
14    /**
15    We use the face values of α to weight the gradients of the
16    5-points Laplacian operator. We get the relaxation function. */
17
18    foreach_level_or_leaf (l) {
19      double n = - sq(Delta)*b[], d = - lambda[]*sq(Delta);
20      foreach_dimension() {
21        n += alpha.x[1]*a[1] + alpha.x[]*a[-1];
22        d += alpha.x[1] + alpha.x[];//此为实现上文公式中的离散格式
23      }
24  #if EMBED
25      if (p->embed_flux) {
26        double c, e = p->embed_flux (point, a, alpha, &c);
27        n -= c*sq(Delta);
28        d += e*sq(Delta);
29      }
30      if (!d)
31        c[] = b[] = 0.;
32      else
33  #endif // EMBED
34        c[] = n/d;//计算该迭代层的 da
35    }
36
37    /**
38    For weighted Jacobi we under-relax with a weight of 2/3. */
39
40  #if JACOBI
41    foreach_level_or_leaf (l)
42      a[] = (a[] + 2.*c[])/3.;
43  #endif
44
45  #if TRASH
46    scalar a1[];
47    foreach_level_or_leaf (l)
48      a1[] = a[];
49    trash ({a});
```

```
  foreach_level_or_leaf (l)
    a[] = a1[];
#endif
}
```

接下来则是 residual 函数，其目的是求解给定值的残差 R，即：

$$R = \mathscr{L}(\delta a) = b - \mathscr{L}(a) = b - \lambda a - \nabla \cdot (\alpha \nabla a) \tag{15}$$

输入值即为上式中的 b, a，具体代码如下：

```
static double residual (scalar * al, scalar * bl, scalar * resl, void * data)
{
  scalar a = al[0], b = bl[0], res = resl[0];
  struct Poisson * p = (struct Poisson *) data;
  (const) face vector alpha = p->alpha;
  (const) scalar lambda = p->lambda;
  double maxres = 0.;
#if TREE
  //此为 2 阶精度，其离散格式为
  ↪ $\frac{(\alpha_{i+\frac{1}{2},j}-\alpha_{i-\frac{1}{2},j})\frac{a_{i,j}-a_{i-1,j}}{\Delta}}{\Delta} + \frac{(\alpha_{i,j+\frac{1}{2}}-\alpha_{i,j-\frac{1}{2}})\Delta\frac{a_{i,j}-a_{i,j-1}}{\Delta}}{\Delta} + \lambda a_{i,j} - \gamma = b$
  face vector g[];
  foreach_face()
    g.x[] = alpha.x[]*face_gradient_x (a, 0);
  foreach (reduction(max:maxres), nowarning) {
    res[] = b[] - lambda[]*a[];
    foreach_dimension()
      res[] -= (g.x[1] - g.x[])/Delta;
#if EMBED
    if (p->embed_flux) {
      double c, e = p->embed_flux (point, a, alpha, &c);
      res[] += c - e*a[];
    }
#endif // EMBED
    if (fabs (res[]) > maxres)
      maxres = fabs (res[]);
  }
#else // !TREE
  /* "naive" discretisation (only 1st order on trees) */
```

```
  //此为一阶精度格式，离散格式为
  ↪ $\frac{\alpha_{i+\frac{1}{2},j}\frac{a_{i+1,j}-a_{i,j}}{\Delta}-\alpha_{i-\frac{1}{2},j}\frac{a_{i,j}-a_{i-1,j}}{\Delta}}{\Delta}+\frac{\alpha_{i,j+\frac{1}{2}}\frac{a_{i,j+1}-a_{i,j}}{\Delta}-\alpha_{i,j-\frac{1}{2}}\frac{a_{i,j}-a_{i,j-1}}{\Delta}}{\Delta}+\lambda a_{i,j}-\gamma=b$
  foreach (reduction(max:maxres), nowarning) {
    res[] = b[] - lambda[]*a[];
    foreach_dimension()
      res[] += (alpha.x[0]*face_gradient_x (a, 0) -
        alpha.x[1]*face_gradient_x (a, 1))/Delta;
#if EMBED
    if (p->embed_flux) {
      double c, e = p->embed_flux (point, a, alpha, &c);
      res[] += c - e*a[];
    }
#endif // EMBED
    if (fabs (res[]) > maxres)
      maxres = fabs (res[]);//选取计算域中最大的残差返回
  }
#endif // !TREE
  return maxres;
}
```

3.5 具体使用示例

最终提供一个能够被广泛运用的函数调用模板，求解的方程为

$$\nabla \cdot (\alpha \nabla a) + \lambda a = b \tag{16}$$

模板函数返回值类型为 $mgstats$，输入值类型为 $structPoisson$，调用 $mg\,cycle$，$mg\,solve$ 等函数

```
mgstats poisson (struct Poisson p)
{
  //如果 α,λ 没有设置，则默认其为单位场
  if (!p.alpha.x.i)
    p.alpha = unityf;
  if (!p.lambda.i)
    p.lambda = zeroc;

  //将这两个参数赋值到每一层级网格中
  face vector alpha = p.alpha;
  scalar lambda = p.lambda;
```

```
  restriction ({alpha,lambda});

  //设置残差容忍，该常数是跳出循环的判定参数，如果没有设置，则使用默认参数值

  double defaultol = TOLERANCE;
  if (p.tolerance)
    TOLERANCE = p.tolerance;

  scalar a = p.a, b = p.b;
#if EMBED
  if (!p.embed_flux && a.boundary[embed] != symmetry)
    p.embed_flux = embed_flux;
#endif // EMBED
  mgstats s = mg_solve ({a}, {b}, residual, relax,
            &p, p.nrelax, p.res, minlevel = max(1, p.minlevel));//注释：多重网
             ↪ 格法中将残差值插值到最小网格层 minlevel，默认 minlevel=1

  /**
  We restore the default. */

  if (p.tolerance)
    TOLERANCE = defaultol;

  return s;
}
```

4. 附录：Basilisk 网格生成与插值

本章基于 [1] 简要阐述 Basilisk 网格生成与插值的基本规则。

4.1 leaf cell

众所周知，Basilisk 细化网格的方式为对网格进行四分或者八分，那么就说明除最开始的 0 级别网格外，每一个网格单元都有其母网格，且每一个网格单元都可以四分或八分成为母网格，而 *leaf cell* 指的就是**不存在子网格的网格单元**，即在这一区域网格细化的终点。

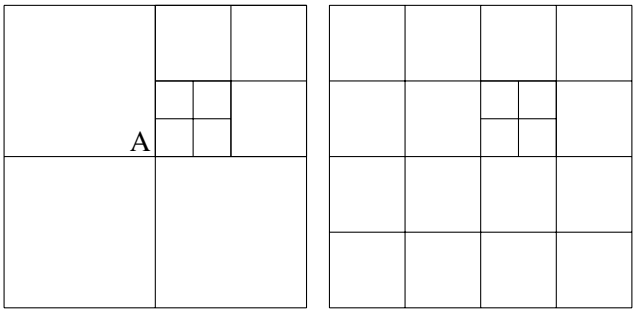
4.2 网格排布规律

Basilisk 中网格严格按照层级递进的规则进行细化，即相邻的网格只可能网格层级相同或相差 1，不存在越级网格相邻。示例如下：

```
#include "grid/quadtree.h"
#include "run.h"
#include "maxruntime.h"

#define MAXTIME 10

scalar f[];
scalar f1[];
scalar * list = {f, f1};
int main(){
^^IL0 = 16;
^^IX0 = Y0 = 0;
  init_grid (2); // Initialize a 2 x 2 grid
  origin(X0,Y0);
  run();
}

event initial(t=0)
{
  refine ((x > 8) && (y > 8) && (level < 2)); // Refine to top right corner
  refine ((x > 8) && (x < 12) && (y > 8) && (y < 12) && (level < 3)); //
  ↪ Refine to top right corner
  //unrefine ((x < 8) && (y < 8) && level >= 1); // Coarsen the bottom left
  ↪ corner
}

event test(t = 0)
{
  int i = 0;
  foreach()
  {
    for(scalar s in list)
    foreach_blockf(s)
    {
      s[] = i;
    }
    i++;
```

```
  }
}

event end(t = 10)
{}
```

如果严格按照上述代码中的命令进行网格加密，则相应效果应该如3(a)



(a) 理论上的网格加密　　(b) 实际上出现的网格加密

图 3: 网格加密形式

原因是 Basilisk 判断到在3(a)A 处出现了相邻网格层级“越迁”的现象，是故将所涉及的周围网格全部重新加密。

4.3 网格点插值与平均

公式12表明计算相应数值需要等距网格中心点处的数值，而由于网格之间可能存在一级的错位，则需要插值或平均得到相应位置的拟合数据再进行计算，如图4

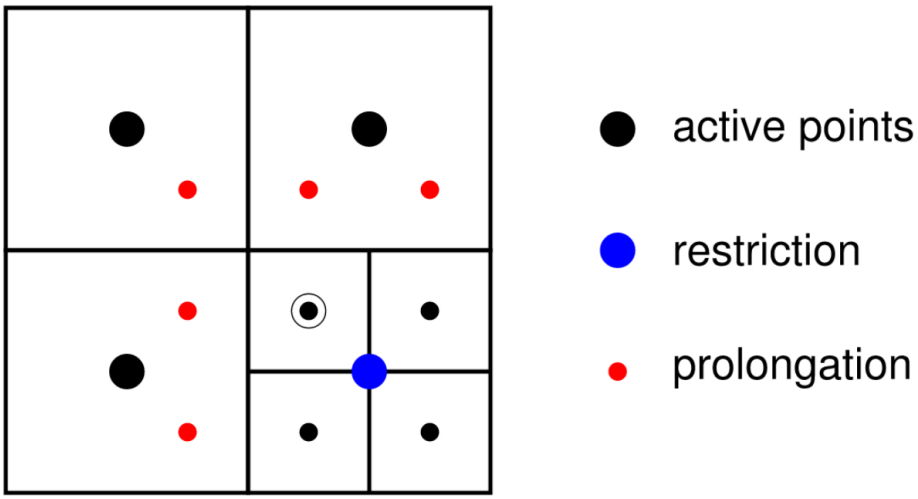


图 4: 网格插值与平均计算示意图，截取自 [1]

如果要求解目标画圈黑点上的相应 $relax$ 函数值，则需要周边同等级数据点的参与，其中

包含并不存在的红色数据点，而想要使用双线性插值求解红色数据点，则需要包围数据点的母网格中心数据，即图中三个黑色点与一个蓝色点。

其中蓝色数据点并不存在，需要使用其子网格上四个数据进行平均得到，从而求取所需的红色数据点。获得相应数据后再经由12计算目标网格数据值即可。

参考文献

[1] Stéphane Popinet. “A quadtree-adaptive multigrid solver for the Serre–Green–Naghdi equations”. In: *Journal of Computational Physics* 302 (2015), pp. 336–358.

[2] Stéphane Popinet. “Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries”. In: *Journal of computational physics* 190.2 (2003), pp. 572–600.