


从 0 开始编写 FEM 程序

By: Zhu YunPeng 



MATLAB 版

日期: 2025 年 2 月 5 日





目录

1 简介	3
2 预备程序	3
2.1 实现 Gauss-Legendre 求积公式	3
2.2 实现参考元上的局部基函数	4
2.3 实现单元上的局部基函数	5
2.4 实现单元上积分 $\int_{E_n} c \psi_{n\alpha}^{(r)} \psi_{n\beta}^{(s)} dx$	5
2.5 实现单元上积分 $\int_{E_n} f \psi_{n\beta}^{(s)} dx$	6
3 1 维 FEM 程序	6
3.1 实现 A 组装器	6
3.1.1 什么是 Parallel Pool	7
3.1.2 为什么使用 Parallel Pool	7
3.1.3 什么情况下使用 Parallel Pool	8
3.1.4 如何使用 Parallel Pool	8
3.1.5 parfor 循环中的各种变量	10
3.1.6 其他材料	12
3.1.7 实现 A 组装器	12
3.2 实现 b 组装器	15
3.3 整合各信息矩阵以及边界条件	16
3.4 实现误差计算	16
3.4.1 实现单元上点的评估	16
3.4.2 实现任意点的评估 (可选)	17
3.4.3 实现获取单元上的高斯点	18
3.4.4 实现各范数误差	18
3.5 实现可视化函数	20
3.6 实现 1 维 FEM	21
3.7 1 维 FEM 程序依赖图	22

4	2 维 FEM 程序	22
4.1	实现两点式直线方程	22
4.2	实现二维线积分	23
4.3	实现三角单元上的 Gauss-Legendre 求积公式	24
4.4	实现信息矩阵整合函数	26
4.4.1	了解坐标-索引关系	26
4.4.2	实现 T 阵	26
4.4.3	实现 Tb 阵	28
4.4.4	实现网格边界信息矩阵	30
4.4.5	实现有限元边界信息矩阵	31
4.4.6	确保有限元边界信息矩阵在交点处的正确性	31
4.4.7	实现信息矩阵整合函数	32
4.5	更新参考元上的基函数	33
4.6	更新单元上的基函数	34
4.7	实现 2 维 FEM	35
4.8	2 维 FEM 程序依赖图	36

1 简介

这份文档旨在为想亲手实现有限元方法的同学提供实用指南. 正如标题所言——从 0 开始编写 FEM 程序, 你只需要准备一台电脑和 MATLAB 2021(或更新版本), 剩下的跟着教程走即可. 如果在阅读过程中发现任何表述不清之处, 欢迎随时提出疑问.

特别感谢赛博助手们—— deepseek、 OpenAI ChatGPT 4.0 的帮助, 它们完成了约占总工作量 20% 的工作. 具体来说  deepseek 负责把将干瘪的初稿转化为生动的讲解 (贡献 15%), 没有它本文的可读性将大打折扣;  OpenAI ChatGPT 4.0 负责在作者为优化算法而冥思苦想时提供了一大堆“看似有用但实则需要魔改”的代码灵感.

2 预备程序

2.1 实现 Gauss-Legendre 求积公式

首先我们需要编写计算 n 阶 Gauss-Legendre 积分节点与权重的子函数 `get_gauss_point_weight`. 这个函数的核心作用是预计算积分参数, 方便后续直接调用. 虽然也可以每次积分时实时计算参数, 但那样会显著增加计算开销——除非你需要频繁调整积分阶数, 否则预计算方案显然更高效.

其数学原理基于 Legendre 多项式的性质: 对于 n 阶求积公式, 节点 x_k 是 $P_{n+1}(x)$ 的零点, 对应权重 A_k 由下式确定:

$$P_{n+1}(x_k) = 0, A_k = \frac{2}{(1 - x_k^2)[P'_{n+1}(x_k)]^2}.$$

其中 Legendre 多项式的显式表达式为:

$$P_{n+1}(x) = \frac{1}{(n+1)!2^{n+1}} \frac{d^{n+1}}{dx^{n+1}} (x^2 - 1)^{n+1}.$$

Listing 1: `get_gauss_point_weight`

```
1 % n 为求积阶数
2 function [point,weight] = get_gauss_point_weight(n)
3 syms x;
4 N = 10;% 有效数字位数
5 Poly = diff((x.^2 - 1).^(n + 1),x,n + 1) ./ (factorial(n + 1) .* 2 .^(n + 1));
6 Poly_d = matlabFunction(diff(Poly,x));
7 point = vpa(solve(Poly),N);
8 weight = vpa(2 ./ ((1 - point.^2).*(Poly_d(point).^2)),N);
9 end
```

调用 `get_gauss_point_weight` 得到对应求积节点与系数后, 接着实现一维 Gauss 积分函数 `gauss_1d`, 其核心思想是通过仿射变换将标准区间 $[-1, 1]$ 映射到任意区间 $[a, b]$:

$$\int_a^b f(x)dx \approx \frac{b-a}{2} \sum_{k=0}^n A_k f\left(\frac{b-a}{2}x_k + \frac{b+a}{2}\right).$$

Listing 2: `gauss_1d`

```
1 % func need to be a function handle which can feed in x.
2 function result = gauss_1d(func,vertices)
3 [xmax,xmin] = deal(max(vertices),min(vertices));
4 gauss_points = [0.9602898565,-0.9602898565,0.7966664774,-0.7966664774,...
5 0.5255324099,-0.5255324099,0.1834346425,-0.1834346425];
6 gauss_weights = [0.1012285363,0.1012285363,0.2223810345,0.2223810345,...
7 0.3137066459,0.3137066459,0.3626837834,0.3626837834];
```

```

8 mapped_gauss_points = (xmax - xmin) ./ 2 .* gauss_points + (xmax + xmin) ./ 2;
9 result = (xmax - xmin) ./ 2 * gauss_weights * func(mapped_gauss_points)';
10 end

```

我们给 `get_gauss_point_weight` 做了两个入参 `func, vertices`, 其中 `func` 只需要是一个能够接受 x 为入参的函数即可, 并不关心具体是什么函数. `vertices` 代表求积区间顶点, 一般是以向量的形式给出, 如 $[x_1, x_2]$, 考虑到不同的局部编号带来的差异, 函数会自动识别区间左右端点.

2.2 实现参考元上的局部基函数

完成积分模块后, 我们着手构建 FEM 的核心基石——参考元局部基函数 `basis_ref_function`. 基函数的高效实现是 FEM 程序性能优化中至关重要的一环. 虽然 MATLAB 符号工具箱能够自动处理求导运算 (从而节省手工推导时间), 但其运行机制决定了每次调用都会重复执行符号解析、微分运算和数值转换等操作. 当网格规模达到一定程度时, 这种反复产生的开销会导致计算时间呈指数级增长. 因此, 我们决定采取“以空间换时间”的策略——多写几行代码, 换来节省数十小时的运行时间. 这就是所谓的“打表法”: 本质上是提前准备好所有答案的小抄, 将所有可能的基函数及其导数一并写好并存储在程序中. 使用时只需直接查表调用, 就如同在考试时翻小抄一样高效. 虽然现在看起来像是在写“流水账”, 但当程序真正运行起来时, 你会为这个决定点赞——毕竟, 谁也不想一整天盯着“MATLAB 正忙”看一整天对吧?

Listing 3: `basis_ref_function`

```

1 function result = basis_ref_function(x, y, dim, type, index, dx, dy)
2 %% dim = 1 type = 1 1D_Linear FE
3 if dim == 1
4     if type == 1
5         if index == 1
6             if dx == 0
7                 result = 1 - x;
8             elseif dx == 1
9                 result = - 1;
10            elseif dx >= 2
11                result = 0;
12            end
13        end
14        if index == 2
15            if dx == 0
16                result = x;
17            elseif dx == 1
18                result = 1;
19            elseif dx >= 2
20                result = 0;
21            end
22        end
23    end
24 end
25 end

```

这里特别解释一下入参 `index`, 它代表的是选取单元上第 `index` 个局部编号指定的有限元节点上的局部基函数, 所以在程序中的 `index` 上的局部基函数得跟局部编号的局部基函数一致.

例如, 假设在单元 $[0, \frac{1}{2}, 1]$ 上, 局部编号是 1, 3, 2, 而 $f(x), g(x), h(x)$, 分别对应 $x_1 = 0, x_3 = \frac{1}{2}, x_2 = 1$ 的局部基函数. 在程序中它们就分别对应 if 条件里的 `index = 1, 3, 2`, 即 `index = 1 \rightarrow f(x), index = 2 \rightarrow h(x), index = 3 \rightarrow g(x)`.

最后,我们为之后的 2 维 FEM 留下了接口——y dy dim, 虽然它们现在还在摸鱼, 但是之后会让它们打工的...

2.3 实现单元上的局部基函数

有了参考单元上的局部基函数, 我们就可以通过仿射变换映射到任意的单元上, 对于 1 维问题来说, 仿射变换为:

$$\hat{x} = \frac{x - x_n}{h},$$

$$\hat{\psi}_1(\hat{x}) = 1 - \hat{x},$$

$$\hat{\psi}_2(\hat{x}) = \hat{x}.$$

Listing 4: basis_local_function

```
1 function result = basis_local_function(x, y, dim, type, index, dx, dy, vertices)
2 %% dim = 1 1D_Linear FE && 1D_Quadratic FE
3 % vertices = [x1,x2]
4 if dim == 1
5     h = max(vertices) - min(vertices);
6     x_hat = (x - min(vertices)) ./ h;
7     if dx == 0
8         result = basis_ref_function(x_hat, y, dim, type, index, dx, dy);
9     elseif dx >= 1
10        result = basis_ref_function(x_hat, y, dim, type, index, dx, dy) ./ h; % d2x_hat/dx^2 = 0
11    end
12 end
13 end
```

这里因为 $\left(\frac{d\hat{x}}{dx}\right)^{(k)} = 0, (k \geq 1)$, 所以当函数请求 $dx \geq 1$ 的局部基函数时, 由牛顿莱布尼茨公式:

$$\hat{\psi}(\hat{x})^{(n+1)} = \left(\frac{d\hat{\psi}}{d\hat{x}} \frac{d\hat{x}}{dx}\right)^{(n)} = \sum_{k=0}^n \binom{n}{k} \left(\frac{d\hat{\psi}}{d\hat{x}}\right)^{(n-k)} \left(\frac{d\hat{x}}{dx}\right)^{(k)} = \left(\frac{d\hat{\psi}}{d\hat{x}}\right)^{(n)} \frac{d\hat{x}}{dx} = \frac{1}{h} \frac{d^{n+1}\hat{\psi}}{d\hat{x}^{n+1}}, n \geq 1,$$

$$\hat{\psi}'(\hat{x}) = \frac{d\hat{\psi}}{d\hat{x}} \frac{d\hat{x}}{dx} = \frac{1}{h} \frac{d\hat{\psi}}{d\hat{x}}.$$

只需要返回 $\frac{1}{h} \frac{d^{n+1}\hat{\psi}}{d\hat{x}^{n+1}}, n \geq 0$ 即可.

2.4 实现单元上积分 $\int_{E_n} c\psi_{n\alpha}^{(r)}\psi_{n\beta}^{(s)}dx$

这一函数是刚度矩阵组装的核心引擎, 有了之前我们实现的 3 个程序, 这一程序就只是把它们排列组合起来:

Listing 5: get_local_integral_A

```
1 % compute each integral
2 % basis_trial/test = [dim type]
3 function result = get_local_integral_A(int_type, alpha, beta, basis_trial, basis_test, func_c,
4     trial_dx, trial_dy, test_dx, test_dy, vertices)
5 % do 1D integral, vertices = [x1,x2]
6 if int_type == 1
7     int_func = @(x) func_c(x, [])...
8         .* basis_local_function(x, [], basis_trial(1), basis_trial(2), alpha, trial_dx,
9         trial_dy, vertices)...
10        .* basis_local_function(x, [], basis_test(1), basis_test(2), beta, test_dx, test_dy
11        , vertices);
```

```

10     result = gauss_1d(int_func, vertices);
11 end
12 end

```

这里的入参 *basis_trial, basis_test* 其实就是 *trial* 函数与 *test* 函数的维数 *dim* 和类型 *type* 拼成的向量。

2.5 实现单元上积分 $\int_{E_n} f \psi_{n\beta}^{(s)} dx$

作为载荷向量组装的核心组件, 其实现与刚度积分共享设计框架, 我们依葫芦画瓢, 稍加更改:

Listing 6: get_local_integral_b

```

1 % compute each integral
2 % basis_trial/test = [dim type]
3 function result = get_local_integral_b(int_type, beta, basis_test, func_f, test_dx, test_dy,
    vertices)
4
5 % do 1D integral, vertices = [x1,x2]
6 if int_type == 1
7     int_func = @(x) func_f(x, [])...
8         .* basis_local_function(x, [], basis_test(1), basis_test(2), beta, test_dx, test_dy
    , vertices);
9     result = gauss_1d(int_func, vertices);
10 end
11 end

```

3 1 维 FEM 程序

3.1 实现 *A* 组装器

有了上面的预备程序, 我们紧接着可以实现 FEM 程序中最重要的一环——矩阵组装器. 如果说前面的基函数是砖块, 这里就是整个摩天大楼的钢结构焊接现场. 矩阵组装器是 FEM 程序中最关键的核心模块, 其实现的优劣直接决定了整个程序的运行效率.

根据 Chapter_1, PPT 上 63 页的伪代码, 我们很容易能写出下面初代组装器的代码, 它完美诠释了什么是“教科书级”的写法——就像用绣花针雕清明上河图, 虽然严谨但实在慢得让人心碎:

Listing 7: Ver1.0-assembler_A

```

1 % Nm is finite elements number
2 % basis_trial/test = [dim type]
3 % func_c need to be a function handle which can feed in (x,y).
4 function A = assembler_A(...)
5 A = sparse(Nf_test, Nf_trial);
6 for ii = 1:Nm
7     S = zeros(Nb_test, Nb_trial); % 初始化单元刚度矩阵
8     vertices = P(:, T(1:1:size(T, 1), ii)); % 单元顶点矩阵
9     for alpha = 1:Nb_trial % 计算单元刚度矩阵
10         for beta = 1:Nb_test
11             S(beta, alpha) = get_local_integral_A(...);
12         end
13     end
14     A(Tb_test(:, ii), Tb_trial(:, ii)) = A(Tb_test(:, ii), Tb_trial(:, ii)) + S;

```

```
15 end
16 end
```

这段程序主要的效率瓶颈在于:

首先,这是一个串行的程序,也就是说 `for` 循环的每个迭代都是依次执行的,更可怕的是,我们嵌套了 3 个 `for` 循环,也就是说每次外部循环的迭代都要等待内部两层循环完全完成,整个过程需要依次跑 $N_m \times Nb_{trial} \times Nb_{test}$ 次迭代.而在 **MATLAB** 中,`for` 循环的效率是最低的,所以应该尽量减少 `for` 循环的使用次数,尽可能多的用向量化和并行化实现.

其次, A 是一个稀疏矩阵,由于稀疏矩阵的存储方式是惰性存储,即一开始不会分配任何内存给 A ,当 A 有非零值插入时,系统再动态地分配存储空间.而在内存中,空闲的内存其实并不是连续的,因此随着 A 的非零值越来越多,重新对 A 分配内存的速度将会越来越慢——这就像是玩拼图游戏:系统要不断寻找合适的内存块来存放新数据,随着非零元素增多,这个找空位的游戏会越来越慢.我们的程序对 A 总共操作了 N_m 次,当网格密度很大时,这会显著限制速度.

最后,考虑到现在的家用电脑一般都会配置一个多核 **CPU**,而恰恰我们组装的过程中,每一个单元的刚度矩阵是彼此独立的,因此应该充分利用所有的 **CPU** 核心,而这一程序却只用 1 个核心干活,其他核心都在围观摸鱼.

在理想情况下,我们只需对矩阵 A 进行一次操作,即实现“一次性构建”.这听起来像是天方夜谭,但实际上通过适当的算法优化和利用 **MATLAB** 的并行计算功能,这一目标是完全可以达到的.因此我们使用 **MATLAB** 自带的 **Parallel Pool** 来实现一个简易的并行组装器.首先我们先来实现矩阵 A 的组装,随后微调该程序来实现 b 的组装.接下来让我们讨论 **MATLAB** 的 **Parallel Pool**,来真正高效的矩阵组装器.

3.1.1 什么是 **Parallel Pool**

Parallel Pool 是 **MATLAB** 提供的一个并行计算工具箱,它可以实现多种需求的并行计算,而对于我们的 **FEM** 程序来说,我们只需要实现 `for` 循环的并行即可,而这也是 **Parallel Pool** 最主要的功能 (`parfor`).**MATLAB** 中的 `parfor` 循环并行执行循环体中的一系列语句.**MATLAB** 客户端发出 `parfor` 命令,并与 **MATLAB** 工作进程协调,在并行池中的工作进程上并行执行循环迭代.客户端将 `parfor` 操作所需的数据发送给工作进程,大部分计算都在工作进程上执行,将结果发送回客户端并进行组合.

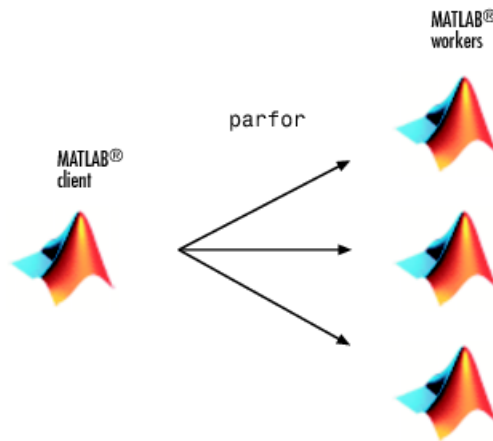


图 1: `parfor` 示意图

3.1.2 为什么使用 **Parallel Pool**

`parfor` 循环可以提供比其类似的 `for` 循环更好的性能,因为多个 **MATLAB** 工作进程可以在同一个循环上同时计算.

`parfor` 循环体的每次执行都是一次迭代.**MATLAB** 工作进程以无特定顺序且彼此独立的方式执行迭代,因为每

次迭代都是独立的, 所以无法保证迭代以任何方式同步, 也没有必要这样做. 如果工作进程数量等于循环迭代次数, 则每个工作进程执行一次循环迭代. 如果迭代次数多于工作进程, 则一些工作进程会执行多次循环迭代; 在这种情况下, 工作进程可能会一次接收多次迭代以减少通信时间.

3.1.3 什么情况下使用 Parallel Pool

当 for 循环开始挑战你的耐心时, parfor 循环就该登场了. 不过就像选择交通工具一样——不是所有路程都适合开跑车, 让我们看看何时该启用 parfor:

1. **计算密集型的长迭代:** 如果某些循环迭代需要较长时间完成, 这时多个工作进程可以并行执行这些任务. 确保总迭代次数超过可用工作进程数量, 以充分利用并行资源.
2. **包含大量简单计算的循环:** parfor 会将循环迭代划分为若干组, 使每个工作进程负责一部分任务, 从而减少总体执行时间.

然而, 在以下情况下使用 parfor 可能效果不佳:

1. **已向量化的代码:** 向量化优化通常能够充分利用 MATLAB 底层库提供的多线程并行特性. 如果代码已经实现了向量化, 强行转为 parfor 反而可能降低性能. 因此, 在这种情况下, 应优先保留向量化结构, 而非改用并行化.(详细了解向量化请参考:https://ww2.mathworks.cn/help/matlab/matlab_prog/vectorization.html).
2. **执行时间较短的循环迭代:** 并行操作本身需要额外的通信和协调开销, 当单次迭代耗时太短时, 这部分开销可能会占据甚至超过实际计算时间.
3. **迭代间存在依赖关系:** parfor 要求每次迭代必须相互独立. 唯一的例外是通过使用归约变量对结果进行累积操作. 关于 parfor 循环中各种变量类型的详细解释, 我们将在后续章节展开讨论.

在决定是否使用 parfor 时, 关键是权衡并行化带来的实际收益与其额外开销. 并行开销主要包括进程间通信、数据协调和传输所需的时间. 当迭代本身执行速度较快时, 这些开销可能会显著影响整体性能. 通常情况下:

1. **适合 parfor:** 需要大量计算且单次迭代耗时较长的循环. 由于计算时间占主导地位, 数据传输的相对开销可以忽略不计.
2. **不建议使用 parfor:** 循环内执行简单操作但次数较多的情况. 此时, 并行化所带来的通信和协调开销可能超过实际计算节省的时间成本.

总之, 选择是否使用 parfor 循环, 要根据具体任务的特点综合考量. 在决定并行化前, 建议先用 tic/toc 对比并行化版本与原始循环的性能差异.

3.1.4 如何使用 Parallel Pool

怎么将 for 循环转换为 parfor 呢, 下面通过一个示例来了解这个过程:

Listing 8: Example_for

```
1 for x = 0:0.1:1
2     for y = 2:10
3         A(y) = A(y-1) + y;
4     end
5 end
```

为了加快代码速度, 尝试将 for 循环转换为 parfor 循环 (也就是把 for 换成 parfor).

Listing 9: Example_parfor

```
1 parfor x = 0:0.1:1
2     parfor y = 2:10
3         A(y) = A(y-1) + y;
```



```
4     end
5 end
```

然而, 这种直接转换的方法往往会遇到问题:

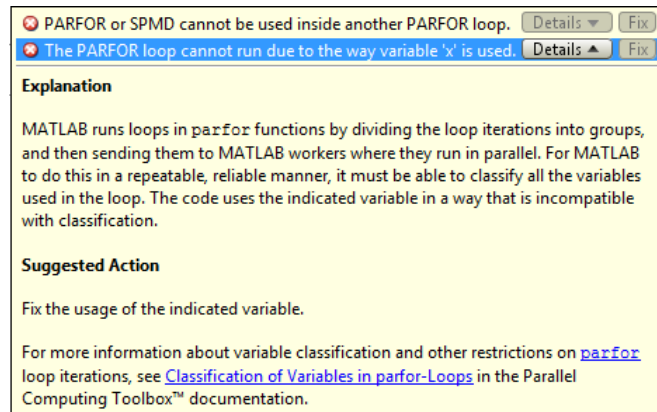


图 2: parfor 报错

Example_parfor 展示了当尝试将 for-loops 转换为 parfor-loops 时常见的问题:

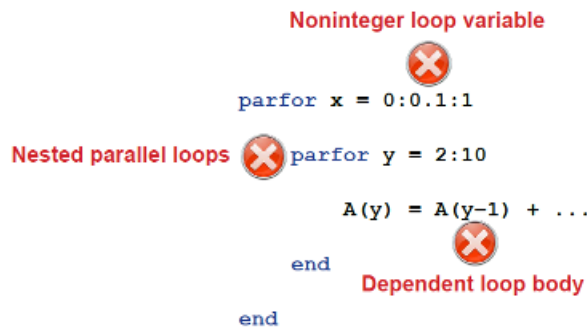


图 3: 常见问题

这些问题同时也是使 parfor 运行起来的必须条件:

1. **循环体依赖性问题:** 因为迭代是以不确定的顺序并行执行的, 所以 parfor 循环的主体必须是独立的, 一次循环迭代不能依赖于前一次迭代. 而在 **Example_parfor** 中:

```
1 A(y) = A(y-1) + y;
```

这个操作依赖于前一次迭代的结果, 因此不能使用 parfor.

2. **嵌套循环问题:** MATLAB 不允许像俄罗斯套娃一样将 parfor 循环嵌套在另一个 parfor 循环内. 不过, 可以调用在另一个 parfor 循环体内使用 parfor 循环的函数. 然而, 这种嵌套的 parfor 循环不会带来任何计算上的好处, 因为所有工作进程都用于并行化最外层循环.
3. **循环变量必须是整数:** parfor 要求循环变量必须是连续增加的整数. 在 **Example_parfor** 中:

```
1 parfor x = 0:0.1:1
```

由于 x 不是整数循环变量, 因此不能在这里使用 parfor. 可以通过将循环变量的值更改为算法所需的整数值来解决此问题.

4. **跳出循环的限制:** 最后, 不能像在 for 循环中那样提前跳出 parfor 循环——你无法通过 break 或 return 让所有工作进程同步停止. 因为每个工作进程独立运行, 如同各自封闭的实验室. 当某个进程触发 break 时, 其他进程仍在继续实验, 导致结果不可预测.

当然, 我们还会遇到其他问题, 接下来我们讨论最后一步——parfor 循环中的各种变量.

3.1.5 parfor 循环中的各种变量

parfor 循环体与普通的 for 循环体不同, 变量之间需要遵守一些特定的规则以避免不同进程之间的工作冲突. 下图展示了 parfor 循环中的各种变量, 我们将会一一对此说明:

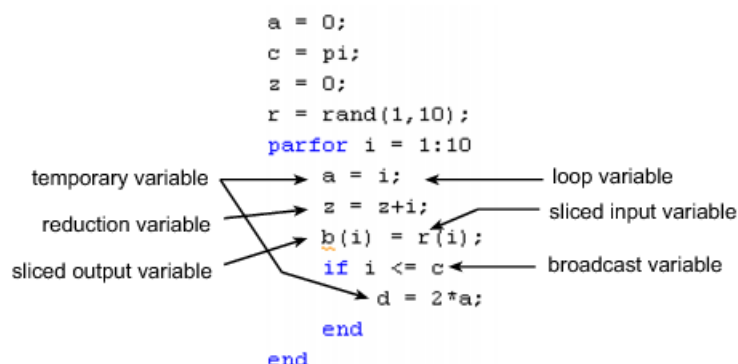


图 4: parfor 循环中的各种变量

临时变量: a 与 d 是典型的“过河拆桥”型变量——每个工作线程在迭代中创建独立副本, 迭代结束时自动销毁. 这种设计确保了线程间的绝对隔离, 但也意味着你无法在 parfor 之后调用它们.

循环变量: i 是并行计算的入口钥匙, 其取值空间被自动划分为互不重叠的子区间分配给各线程. 而当嵌套常规 for 循环时:

```
1 parfor i = 1:1:5
2     for j = 1:1:5
3         % do something
4     end
5 end
```

此时 i 承担战略调度角色 (循环变量必须为 parfor 变量), 而 j 仅执行战术任务 (降级为普通索引——嵌套的 for 循环索引变量). 这种层级关系为后续的分片规则埋下重要伏笔.

广播变量: c 扮演着中央广播台的角色——在 parfor 启动前就已存在 (如 P, T 矩阵), 所有线程共享其只读副本. 这就像是办事大厅的告示一样——普通人只能看不能写——不能在 parfor 循环中尝试更改广播变量.

规约变量: z 通过运算符的交换律特性 (如加法) 实现量子纠缠般的跨线程同步. 虽然表面看来 $z = z + i$ 与执行顺序相关, 但交换律保障了终值 $\sum_{i=1}^{10} i$ 始终等于 55. 但若使用非交换操作 (如矩阵连乘), 程序会出现不可预知的结果——就连 CPU 自己都不知道自己的运算顺序.

最后, 我们来介绍规则最复杂同时也最难理解的变量——**切片变量:** b, r , 这其中 b 可以细分为切片 (输出) 变量, r 可以细分为切片 (输入) 变量. 实际上, 切片变量的规则相当复杂 (<https://ww2.mathworks.cn/help/parallel-computing/sliced-variable.html>), 但是在这里我们会用通俗易懂的方式讲述关键的部份, 其它细节如有需要可以自行查看官方文档. **切片变量是指数组的不同切片由循环的不同迭代进行操作, 那什么叫做切片呢?** 一个被正确切片的数组, 是被下列规则索引的数组:

1. 索引中必须包含一个为 i 的线性组合 ($i \pm k, k$ 为编译期整数常量——标量整数常量或简单 (非索引) 整数广播变量.)
2. 其它的索引表达式仅限于一个常数 (包括正整数常量、一个简单 (非索引) 广播变量、一个嵌套的 for 循环索引变量或者 end) 或者冒号 (:).

这个看似复杂的规则实际上在贯彻一个简单理念: **确保每个线程操作数组的独立分区, 就像魔方高手同时转动不同面而不会引发冲突.** 下面我们用几张图解释不同维度下各种正确的和错误的切片都长的什么样子 (注意, 不正确的切片形式可能千奇百怪, 但是正确的形式只有几种):

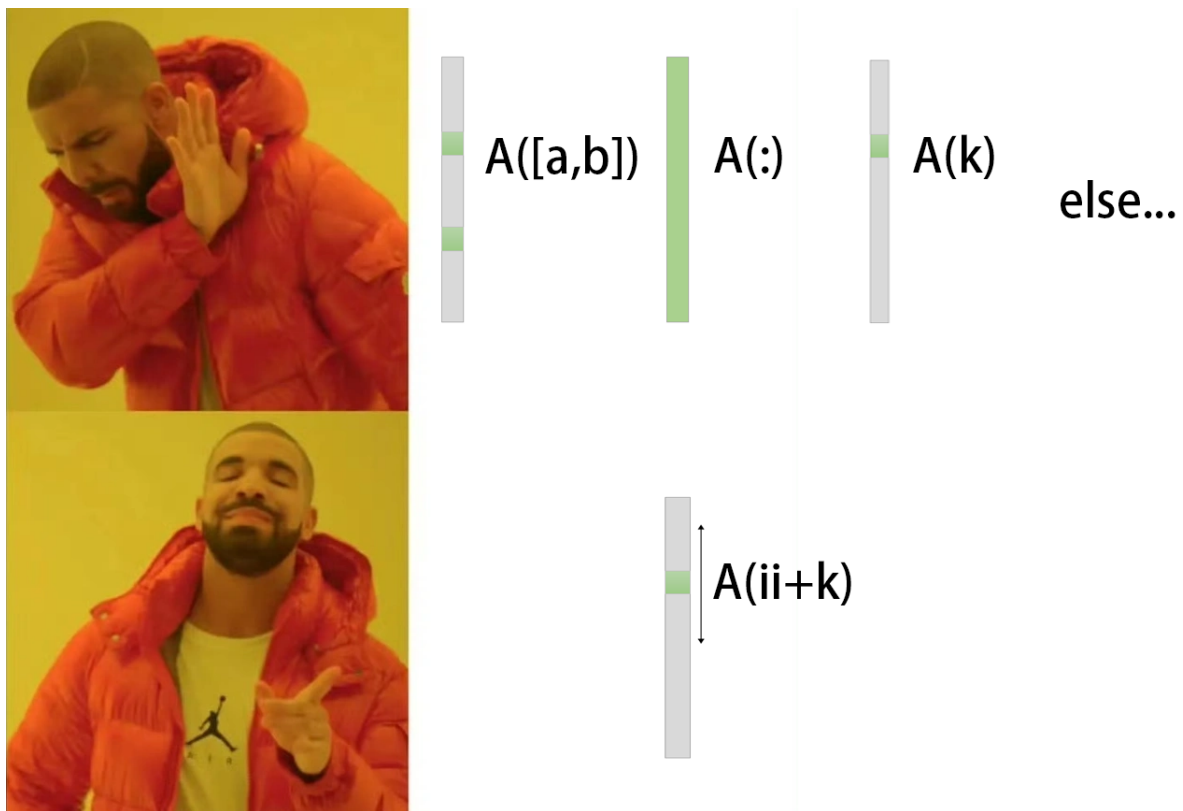


图 5: 1 维的各种切片

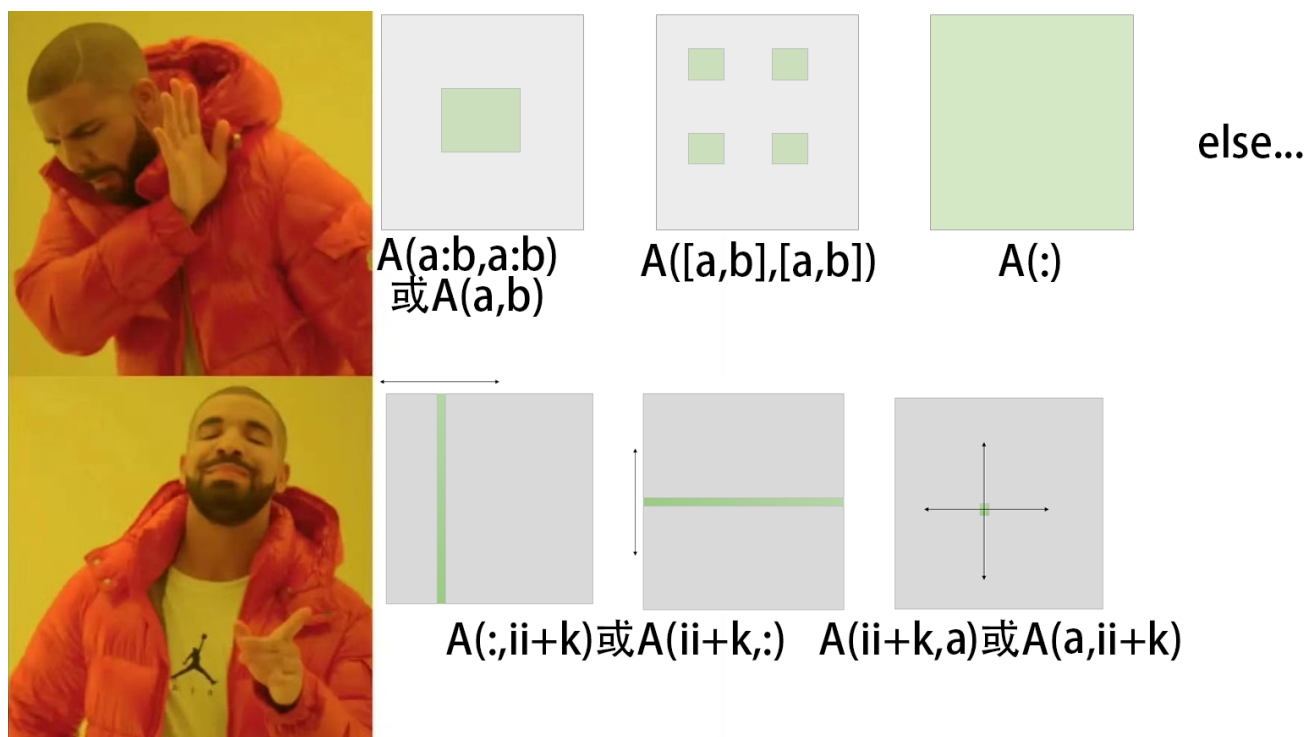


图 6: 2 维的各种切片

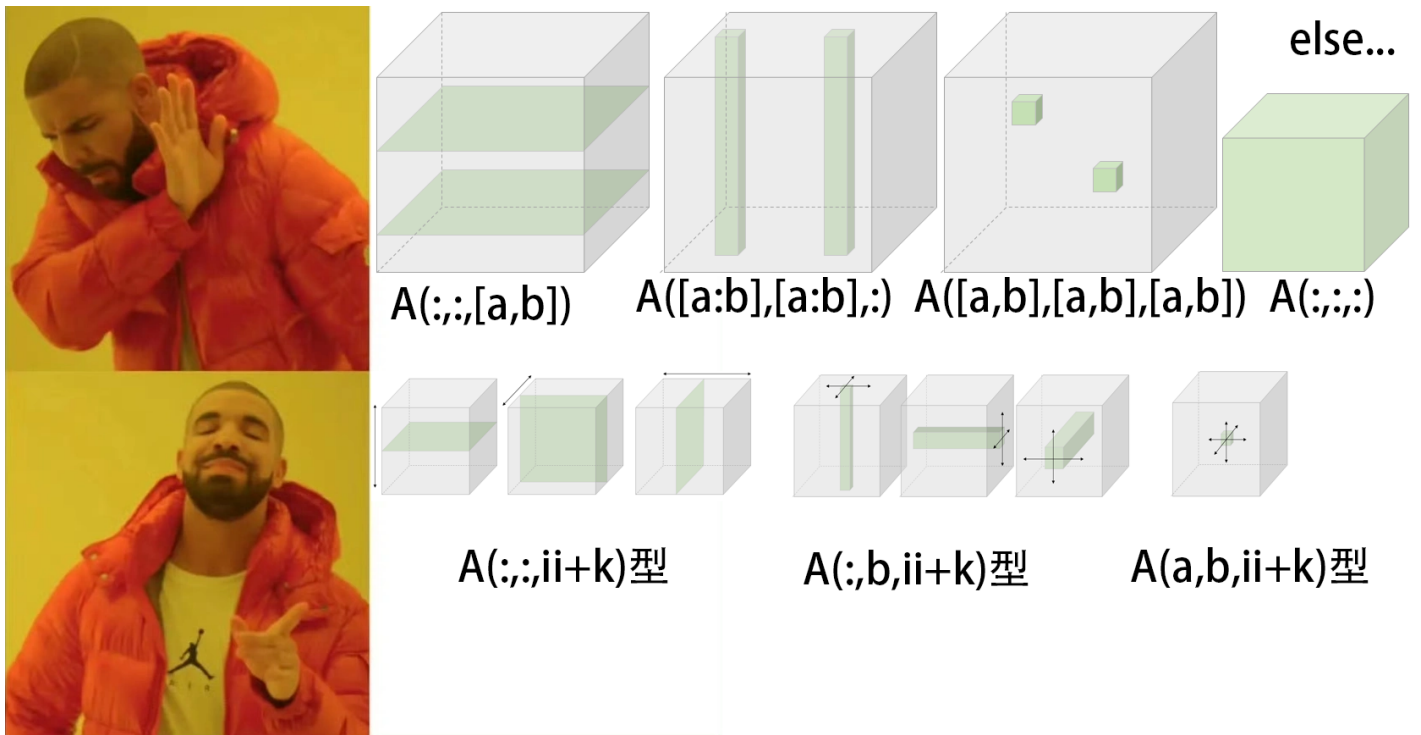


图 7: 3 维的各种切片

3.1.6 其他材料

在使用 `parfor` 时, 难免会遇到各种各样的问题, 这一节给出了 MATLAB 官方的各类文档供需要时查看:

1. 关于并行计算:https://ww2.mathworks.cn/help/overview/parallel-computing.html?s_tid=hc_product_group_bc
2. 关于 Parallel Pool:https://ww2.mathworks.cn/help/parallel-computing/index.html?s_tid=CRUX_lftnav
3. 关于使用 `parfor`:<https://ww2.mathworks.cn/help/parallel-computing/interactively-run-a-loop-in-parallel.html>
4. 关于何时使用 `parfor`:<https://ww2.mathworks.cn/help/parallel-computing/decide-when-to-use-parfor.html>
5. 关于如何使用 `parfor`:<https://ww2.mathworks.cn/help/parallel-computing/convert-for-loops-into-parfor.html>
6. 关于确保 `parfor` 循环体内是独立的:<https://ww2.mathworks.cn/help/parallel-computing/ensure-that-parfor.html>
7. 关于嵌套 `parfor`:<https://ww2.mathworks.cn/help/parallel-computing/nested-parfor-loops-and-for-loops.html>
8. 关于 `parfor` 中的变量故障:<https://ww2.mathworks.cn/help/parallel-computing/troubleshoot-variables-in-parfor.html>
9. 关于透明度故障:<https://ww2.mathworks.cn/help/parallel-computing/transparency.html>

3.1.7 实现 A 组装器

终于, 我们可以着手改进之前的程序了, 首次尝试直接将串行代码中的 `for` 换成 `parfor`:

Listing 10: Ver2.0-assembler_A

```
1 % Nm is finite elements number
2 % basis_trial/test = [dim type]
```

```

3 % func_c need to be a function handle which can feed in (x,y).
4 function A = assembler_A(...)
5 A = sparse(Nf_test, Nf_trial);
6 parfor ii = 1:Nm
7     S = zeros(Nb_test, Nb_trial); % 初始化单元刚度矩阵
8     vertices = P(:, T(1:1:size(T, 1), ii)); % 单元顶点矩阵
9     for alpha = 1:Nb_trial % 计算单元刚度矩阵
10         for beta = 1:Nb_test
11             S(beta, alpha) = get_local_integral_A(...);
12         end
13     end
14     A(Tb_test(:,ii), Tb_trial(:,ii)) = A(Tb_test(:,ii), Tb_trial(:,ii)) + S;
15 end
16 end

```

随后,MATLAB 给了我们当头一棒:

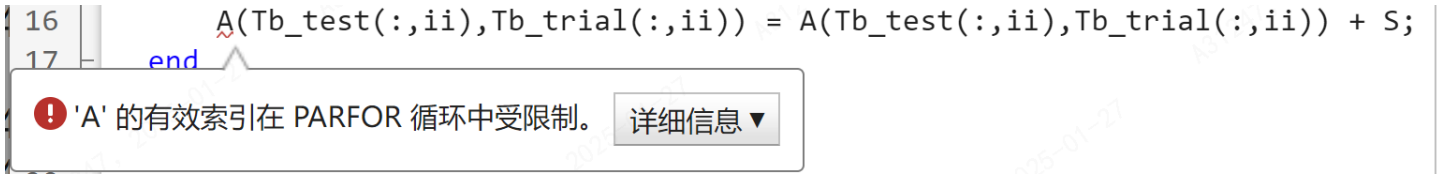


图 8: A 的索引问题

这里 A 的索引有什么问题呢? 注意了: 在我们的程序中,A 并没有被正确的切片:

```

1 A(Tb_test(:,ii), Tb_trial(:,ii))

```

A 的索引实际上是被两个索引了的广播变量索引的, 这实际上是不合法的切片. 根据我们的大原则: 这样的切片并不能保证在任何情况下, 不同的工作进程操作同一数组的不同部份, 因为索引的广播变量可能会有相同的部份. 这也就是为什么 MATLAB 提示:‘A’的有效索引在 PARFOR 循环中受限制——这就像多人同时修改同一 Excel 表格的相同单元格, 必然导致数据混乱, 其本质是:

$$\exists(ii \neq jj), Tb_{test}(:, ii) \cap Tb_{test}(:, jj) \neq \emptyset,$$

或

$$\exists(ii \neq jj), Tb_{trial}(:, ii) \cap Tb_{trial}(:, jj) \neq \emptyset.$$

即不同单元的测试函数/试探函数在全局矩阵中可能共享相同索引. 当然 MATLAB 可不知道实际上到底有没有重复, 它只是简单的阻止不合法的索引而已. 那要如何修改呢? 既然 parfor 中不允许这么索引 A, 那我们只能“曲线救国”——把 A 移到 parfor 外面, 在 parfor 里面记录所有的 S, 再在外面通过普通的 for 循环组装 A, 这就有了我们的第三版程序:

Listing 11: Ver3.0-assembler_A

```

1 % Nm is finite elements number
2 % basis_trial/test = [dim type]
3 % func_c need to be a function handle which can feed in (x,y).
4 function A = assembler_A(int_type, Nf_trial, Nf_test, Nm, P, T, Nb_trial, Nb_test, basis_trial,
    basis_test, Tb_trial, Tb_test, func_c, trial_dx, trial_dy, test_dx, test_dy)
5 A = sparse(Nf_test, Nf_trial);
6 S = cell(1, Nm); % 记录所有的 S
7 parfor ii = 1:Nm
8     S_ii = zeros(Nb_test, Nb_trial); % 初始化单元刚度矩阵

```

```

9     vertices = P(:, T(1:1:size(T, 1), ii)); % 单元顶点矩阵
10    for alpha = 1:Nb_trial % 计算单元刚度矩阵
11        for beta = 1:Nb_test
12            S_ii(beta, alpha) = get_local_integral_A(int_type, alpha, beta, basis_trial, basis_test,
13                func_c, trial_dx, trial_dy, test_dx, test_dy, vertices);
14        end
15    end
16    S{ii} = S_ii;
17    for ii = 1:1:Nm
18        A(Tb_test(:,ii),Tb_trial(:,ii)) = A(Tb_test(:,ii),Tb_trial(:,ii)) + S{ii};
19    end
20 end

```

这段代码虽然实现了并行化, 但仍有优化空间——毕竟每次循环都要操作稀疏矩阵 A 实在不够优雅. 我们注意到 A 本质上是个稀疏矩阵, 而稀疏矩阵的存储精髓就在于只需记住它的三件套: 行坐标、列坐标、对应的值. 因此我们可以尝试一次性打包所有非零元素的信息. 而 MATLAB 的 `sparse` 函数恰好提供了这样的“一键装机”服务, 更妙的是它自带自动合并同类项功能——就像收银台会自动把两盒牛奶合并成数量 2 的条目. 这完美契合了有限元组装时需要累加不同单元贡献的特性.

`S = sparse(i,j,v)` 根据 i 、 j 和 v 三元组生成稀疏矩阵 S , 以便 $S(i(k),j(k)) = v(k)$. $\max(i) \times \max(j)$ 输出矩阵为 $\text{length}(v)$ 个非零值元素分配了空间. 如果输入 i 、 j 和 v 为向量或矩阵, 则它们必须具有相同数量的元素. 参量 v 和/或 i 或 j 其中一个参量可以使标量.

`S = sparse(i,j,v,m,n)` 将 S 的大小指定为 $m \times n$.

图 9: sparse 函数

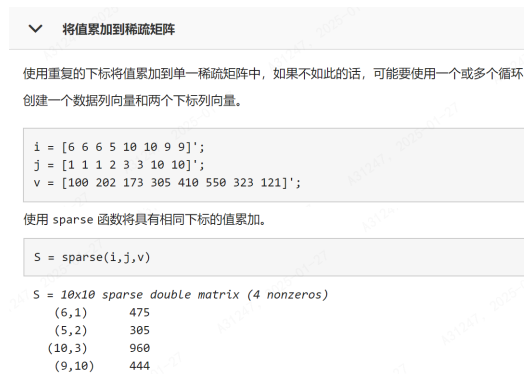


图 10: sparse 索引累加示例

按照这一思路, 我们终于写出了只对 A 操作 1 次的组装器:

Listing 12: Ver4.0-assembler_A

```

1 % Nm is finite elements number
2 % basis_trial/test = [dim type]
3 % func_c need to be a function handle which can feed in (x,y).
4 function A = assembler_A(int_type, Nf_trial, Nf_test, Nm, P, T, Nb_trial, Nb_test, basis_trial,
    basis_test, Tb_trial, Tb_test, func_c, trial_dx, trial_dy, test_dx, test_dy)
5     rows = cell(1, Nm); % 用于存储行索引
6     cols = cell(1, Nm); % 用于存储列索引
7     vals = cell(1, Nm); % 用于存储值
8
9     parfor ii = 1:Nm

```



```

10     S_ii = zeros(Nb_test, Nb_trial); % 初始化单元刚度矩阵
11     vertices = P(:, T(1:1:size(T, 1), ii)); % 单元顶点矩阵
12     for alpha = 1:Nb_trial % 计算单元刚度矩阵
13         for beta = 1:Nb_test
14             S_ii(beta, alpha) = get_local_integral_A(int_type, alpha, beta, basis_trial,
basis_test, func_c, trial_dx, trial_dy, test_dx, test_dy, vertices);
15         end
16     end
17     temp_row = Tb_test(1:Nb_test, ii);
18     temp_col = Tb_trial(1:Nb_trial, ii);
19     [row,col] = meshgrid(temp_row,temp_col);
20     [row,col] = deal(row',col');
21     idx = (S_ii ~= 0); % 找到S_ii中所有非零元素及其对应索引
22     rows{ii} = row(idx);
23     cols{ii} = col(idx);
24     vals{ii} = S_ii(idx);
25 end
26
27 % 将所有非零元素合并到全局稀疏矩阵
28 rows = vertcat(rows{:});
29 cols = vertcat(cols{:});
30 vals = vertcat(vals{:});
31 A = sparse(rows, cols, vals, Nf_test, Nf_trial);
32 end

```

3.2 实现 b 组装器

既然刚度矩阵 A 已经实现了高效组装, 组装载荷向量 b 就轻松多了——毕竟 b 是一个稠密向量, 直接套用组装 A 的第三版稍加调整即可:

Listing 13: assembler_b

```

1 % Nm is finite elements number
2 % basis_trial/test = [dim type]
3 % func_c need to be a function handle which can feed in (x,y).
4 function b = assembler_b(int_type, Nf_test, Nm, P, T, Nb_test, basis_test, Tb_test, func_f, test_dx,
test_dy)
5 b = zeros(Nf_test, 1); % 初始化负载矩阵
6 temp = cell(1,Nm);
7 parfor ii = 1:Nm % 跑遍每个单元
8     S = zeros(Nb_test, 1); % 初始化单元负载矩阵
9     vertices = P(:,T(1:1:size(T,1),ii)); % 单元顶点矩阵
10    for beta = 1:1:Nb_test
11        S(beta,1) = get_local_integral_b(int_type, beta, basis_test, func_f, test_dx, test_dy,
vertices);
12    end
13    temp{ii} = S; % 将单元负载矩阵存入元胞中
14 end
15 for ii = 1:1:Nm
16     b(Tb_test(:, ii),1) = b(Tb_test(:, ii),1) + temp{ii}; % 将单元负载矩阵存入元胞中
17 end
18 end

```


3.3 整合各信息矩阵以及边界条件

由于 1 维的各信息矩阵都十分简单, 我们在此不另行编制子程序, 下面提供一种简单的信息矩阵生成代码供参考:

```
1 b_type = [-1 -1];% 边界条件 -1 -> 'D', -2 -> 'Neu', -3 -> 'Robin'
2 % Neu/Robin = u'+qu=p
3 extra_info = [-1 1];% 为 Neu/Robin 准备的额外信息
4 extra_info_q = [0 0];% 为 Neu/Robin 准备的额外信息(q)
5 extra_info_p = [0 0];% 为 Neu/Robin 准备的额外信息(p)
6
7 %% 构建P,T,Pb,Tb信息矩阵
8 P = left:hx:right;% 网格点坐标矩阵
9 T = [1:1:Nf - 1;2:1:Nf];% 网格点索引矩阵
10 Pb = P;% 有限元点坐标矩阵
11 Tb = T;% 有限元点索引矩阵
12 B_f_edges = [b_type;Tb(1,1),Tb(end,end)];extra_info;extra_info_q;extra_info_p;% 有限元边界信息矩阵
    (第一行是边界类型,第二行是有限元边节点索引,第三/四行是额外信息(如有))
```

Listing 14: boundary_adjust

```
1 % 1D boundary adjuster
2 function [A,b] = boundary_adjust(A, b, Pb, B_info, func_boundary, func_c)
3 for ii = 1:1:2
4     if B_info(1,ii) == -1
5         jj = B_info(2,ii);
6         A(jj,:) = 0;
7         A(jj,jj) = 1;
8         b(jj) = func_boundary(Pb(jj),[]);
9     end
10    if B_info(1,ii) == -2
11        jj = B_info(2,ii);
12        b(jj) = b(jj) + B_info(3,ii) * B_info(5,ii) * func_c(Pb(jj),[]);
13    end
14    if B_info(1,ii) == -3
15        jj = B_info(2,ii);
16        A(jj,jj) = A(jj,jj) + B_info(3,ii) * B_info(4,ii) * func_c(Pb(jj),[]);
17        b(jj) = b(jj) + B_info(3,ii) * B_info(5,ii) * func_c(Pb(jj),[]);
18    end
19 end
20 end
```

3.4 实现误差计算

3.4.1 实现单元上点的评估

在编写误差计算程序时, 评估有限元解在特定点的值是绕不开的任务. 若该点恰好是有限元节点, 问题自然简单——但如果是任意位置的点呢? 我们先聚焦于单个单元内的评估方法:

先设 $\hat{x} \in E_k$, 由有限元解:

$$u_h(x) = \sum_{i=1}^N u_i \phi_i(x).$$

有:

$$u_h(\hat{x}) = \sum_{i=1}^N u_i \phi_i(\hat{x}).$$

而对于全局基函数来说, 有:

$$\phi_i(x_j) = \delta_{ij} = \begin{cases} 1, i = j, \\ 0, i \neq j. \end{cases}$$

故:

$$u_h(\hat{x}) = u_k \phi_k(\hat{x}) = \sum_{i=1}^{Nb_{trial}} u_{ki} \psi_{ki}(\hat{x}).$$

Listing 15: get_element_local_basis_pvalue

```
1 % use to evaluate arbitrary point in one element
2 % WILL NOT CHECK WHETHER THE POINT IN ELEMENT OR NOT.
3 function result = get_element_local_basis_pvalue(uh, x, y, element_index, Nb, P, T, Tb, dim, type,
    dx, dy)
4 uh_local = uh(Tb(:,element_index));
5 temp = zeros([size(x),Nb]);% 存储评估点在各局部基函数上的值
6 for ii = 1:1:Nb% 遍历所有局部基函数
7     temp(:, :, ii) = uh_local(ii) .* basis_local_function(x, y, dim, type, ii, dx, dy, P(:,T(:,
        element_index)));
8 end
9 result = sum(temp,3);
10 end
```

这里我们采用了向量化算法, 通过三维数组同时处理空间坐标和基函数维度, 这种向量化操作比传统循环快得能让咖啡凉透之前就得出结果:

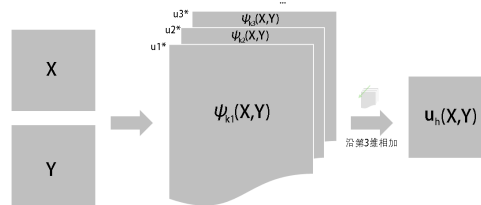


图 11: 向量化算法

3.4.2 实现任意点的评估 (可选)

之前设 $\hat{x} \in E_k$, 但如果不知道 \hat{x} 属于哪一个 E_k 怎么办, 那我们只能无奈地挨家挨户跑遍每个单元来送评估点“回家”: 对于一维来说, 每个单元都是一个区间; 而对于二维, 每个单元都是一个多边形 (利用 MATLAB 自带的判断评估点是否位于某一多边形区域内的函数 `inpolygon`).

Listing 16: get_uh_point

```
1 % point = [x1 x2 ...] in 1D or point = [x1 x2 x3...;y1 y2 y3...] in 2D
2 function result = get_uh_point(uh, point, Nm, Nb, P, T, Tb, dim, type, dx, dy)
3 result = zeros(1,size(point,2));% 存储评估点处的值
4 for ii = 1:1:size(point,2)
5     for jj = 1:1:Nm% 跑遍每个单元
```

```

6         if dim == 1
7             if point(ii) >= P(jj) && point(ii) <= P(jj + 1) % 在 E_jj 上
8                 result(ii) = get_element_local_basis_pvalue(uh, point(ii), [], jj, Nb, P, T, Tb, dim
, type, dx, dy);
9                 break;% 继续下个评估点
10            end
11        else % dim == 2
12            if inpolygon(point(1,ii), point(2,ii), P(1,T(:,jj)), P(2,T(:,jj))) % 判断是否在 E_jj 上
13                result(ii) = get_element_local_basis_pvalue(uh, point(1,ii), point(2,ii), jj, Nb, P,
T, Tb, dim, type, dx, dy);
14                break; % 继续下个评估点
15            end
16        end
17    end
18 end
19 end

```

注意: 这一程序实际上并不会在真正的程序中用到, 因为总是可以通过先遍历单元的办法来规避掉判断评估点是否属于某一个单元的困扰, 但是我们依旧给出这一程序以便某些特殊需求使用。

3.4.3 实现获取单元上的高斯点

这一程序主要是为了后面的 $\|\cdot\|_\infty$ 使用, 当然在单元上随机抽取点也是可以的, 我们采用仿射变换将 $[-1, 1]$ 上的高斯点映到实际单元 $[x_{min}, x_{max}]$ 上, 内嵌的高斯点调用 **get_gauss_point_weight** 即可。

Listing 17: get_element_gauss_point

```

1 % vertices = [x1 x2] in 1D or vertices = [x1 x2 x3;y1 y2 y3] in 2D
2 function mapped_gauss_points = get_element_gauss_point(dim, vertices)
3 gauss_reference_points_1d = [0.9602898565, -0.9602898565, 0.7966664774, ...
4 -0.7966664774, 0.5255324099, -0.5255324099, 0.1834346425, -0.1834346425];
5 if dim == 1
6     [xmax,xmin] = deal(max(vertices),min(vertices));
7     mapped_gauss_points = (xmax - xmin) ./ 2 .* gauss_reference_points_1d + (xmax + xmin) ./ 2;
8 end
9 end

```

3.4.4 实现各范数误差

误差评估系统通过函数组合实现模块化设计: 为方便说明, 我们形式上记:

$$\begin{aligned}
 f &:= \text{get_element_gauss_point}, \\
 w_h &:= \text{get_element_local_basis_pvalue}, \\
 GS &:= \text{gauss_1d}
 \end{aligned}$$

则各范数误差可以表示为:

$$\begin{aligned}
\|u - u_h\|_\infty &= \sup_{x \in I} |u(x) - u_h(x)| = \max_{1 \leq n \leq N} \max_{x_n \leq x \leq x_{n+1}} |u(x) - u_h(x)|, \\
&\approx \max_{1 \leq n \leq N} \max_{x \in f(E_n)} |u(x) - w_h(x, E_n)|, \\
\|u - u_h\|_0 &= \sqrt{\int_I (u(x) - u_h(x))^2 dx} = \sqrt{\sum_{n=1}^N \int_{x_n}^{x_{n+1}} (u(x) - u_h(x))^2 dx}, \\
&\approx \sqrt{\sum_{n=1}^N GS((u(x) - w_h(x, E_n))^2)}, \\
\|u - u_h\|_{H^1_{semi}} &= \sqrt{\int_I (u'(x) - u'_h(x))^2 dx} = \sqrt{\sum_{n=1}^N \int_{x_n}^{x_{n+1}} (u'(x) - u'_h(x))^2 dx}, \\
&\approx \sqrt{\sum_{n=1}^N GS((u'(x) - w_h(x, E_n, dx = 1))^2)}, \\
\|u - u_h\|_{H^1} &= \sqrt{\int_I (u'(x) - u'_h(x))^2 dx + \int_I (u(x) - u_h(x))^2 dx} = \sqrt{\sum_{n=1}^N \int_{x_n}^{x_{n+1}} (u'(x) - u'_h(x))^2 + (u(x) - u_h(x))^2 dx}, \\
&\approx \sqrt{\sum_{n=1}^N GS((u'(x) - w_h(x, E_n, dx = 1))^2) + GS((u(x) - w_h(x, E_n))^2)}.
\end{aligned}$$

Listing 18: err_eval

```

1 % max, L^inf, L^2, H^1(semi), H^1 误差
2 function err = err_eval(real_u, left, right, uh, Nm, Nb, P, T, Pb, Tb, dim, type, hx, err_types)
3 err = hx;
4 F = @(x,y,element_index,dx,dy)(real_u(x,y,dx,dy) - get_element_local_basis_pvalue(uh, x, y,
    element_index, Nb, P, T, Tb, dim, type, dx, dy)).^2;
5 G = @(x,y,element_index)max(abs(real_u(x,y,0,0) - get_element_local_basis_pvalue(uh, x, y,
    element_index, Nb, P, T, Tb, dim, type, 0, 0)));
6 for ii = 1:length(err_types)
7     switch err_types(ii)
8         case "max"% 节点max 误差
9             temp = max(abs(real_u(Pb(1,:),0,0)' - uh));
10        case "L^inf"% L^inf 误差
11            temp_all = zeros(1,Nm);
12            parfor element_index = 1:1:Nm
13                eval_points = get_element_gauss_point(1, P(:,T(:,element_index)));
14                temp_all(element_index) = G(eval_points(1,:),[],element_index);
15            end
16            temp = max(temp_all);
17        case "L^2"% L^2 误差
18            temp = 0;
19            parfor element_index = 1:1:Nm
20                f_1 = @(x)F(x,[],element_index,0,0);
21                sum = gauss_1d(f_1,P(:,T(:,element_index)));
22                temp = temp + sum;
23            end
24            temp = sqrt(temp);
25        case "H^1(semi)"% H^1(semi) 误差
26            temp = 0;

```

```

27         parfor element_index = 1:1:Nm
28             f_1 = @(x)F(x,[],element_index,1,0);
29             sum = gauss_1d(f_1,P(:,T(:,element_index))));
30             temp = temp + sum;
31         end
32         temp = sqrt(temp);
33     case "H^1"% H^1 误差
34         temp = 0;
35         parfor element_index = 1:1:Nm
36             f_1 = @(x)F(x,[],element_index,1,0) + F(x,[],element_index,0,0);
37             sum = gauss_1d(f_1,P(:,T(:,element_index))));
38             temp = temp + sum;
39         end
40         temp = sqrt(temp);
41     case "all"
42         temp = err_eval(real_u, left, right, uh, Nm, Nb, P, T, Pb, Tb, dim, type, hx, ["max","L^
inf","L^2","H^1","H^1(semi)"]);
43         temp = temp(3:end);% 去掉重复的 h
44     otherwise
45         error(['Unsupported error type: ' err_types(ii)]);
46     end
47     err = [err temp];
48 end
49 end

```

3.5 实现可视化函数

算出有限元解 u_h 后, 总得看看它和真实解到底有多像——就像刚拼完乐高要对照说明书检查一样. 这里提供一个开箱即用的绘图程序:

Listing 19: show_eval

```

1 % 可视化函数
2 % compute finite nodes error.
3 % so we feed in point_x point_y as finite coordinate.
4 function show_eval(real_u, uh, Pb, txt)
5 uh_y = uh;
6 % 比较
7 figure;
8 plot(Pb,real_u(Pb,[],0,0),'LineWidth',2);
9 hold on;
10 plot(Pb,uh_y,'--','LineWidth',2);
11 grid on;
12 grid minor;
13 title("Comparsion (" + txt + ")");
14 xlabel('x');
15 ylabel('y');
16 legend('Numerical Solution','Analytic Solution');
17 % 误差分析
18 abs_error = abs(real_u(Pb,[],0,0) - uh_y);
19 figure;
20 plot(Pb,abs_error,'LineWidth',2);

```

```

21 grid on;
22 grid minor;
23 title("Absolute Error (" + txt + ")");
24 xlabel('x');
25 ylabel('Absolute Error');
26 end

```

想要个性化调整？改改线型颜色、调调字体大小什么的就跟刷手机一样简单. 这个基础版相当于给你搭好了画架, 具体怎么上色就看你自己了.

3.6 实现 1 维 FEM

最后, 我们已经成功完成了所有必要的“零件”, 现在就像拼乐高一样, 把之前造好的零件组装成完整的 1 维有限元求解器. 这个主函数就是总装车间: 只需要把所有“零件”拼在一起就可以了:

Listing 20: FEM_1D_Linear_Dirichlet

```

1 function [uh,err] = FEM_1D_Linear_Dirichlet(input_h,fig_flag,err_flag,err_types)
2 %% 输入网格信息
3 left = 0;
4 right = 1;
5 hx = input_h(1);% x 网格步长
6 Nx = (right - left) / hx;% x网格单元数
7 b_type = [-1 -1];% 边界条件 -1 -> 'D', -2 -> 'Neu', -3 -> 'Robin'
8 % Neu/Robin = u'+qu=p
9 extra_info = [-1 1];% 为 Neu/Robin 准备的额外信息
10 extra_info_q = [0 0];% 为 Neu/Robin 准备的额外信息(q)
11 extra_info_p = [0 0];% 为 Neu/Robin 准备的额外信息(p)
12
13
14 %% 输入有限元信息
15 Nf = Nx + 1;% x有限元节点数
16 Nm = Nx;% 网格单元数
17 Nb = 2;% 局部基函数个数
18 dim = 1;% 1D
19 type = 1;% Linear
20
21 %% 构建P,T,Pb,Tb信息矩阵
22 P = left:hx:right;% 网格点坐标矩阵
23 T = [1:1:Nf - 1;2:1:Nf];% 网格点索引矩阵
24 Pb = P;% 有限元点坐标矩阵
25 Tb = T;% 有限元点索引矩阵
26 B_f_edges = [b_type;Tb(1,1),Tb(end,end)];extra_info;extra_info_q;extra_info_p;% 有限元边界信息矩阵
    (第一行是边界类型,第二行是有限元边节点索引,第三/四行是额外信息(如有))
27
28 %% 组装刚度矩阵与荷载矩阵
29 A = assembler_A(1, Nf, Nf, Nm, P, T, Nb, Nb, [dim type], [dim type], Tb, Tb, @func_c, 1, [], 1, []);
30 b = assembler_b(1, Nf, Nm, P, T, Nb, [dim type], Tb, @func_f, 0, []);
31
32 %% 使用边界条件信息矩阵调整边界条件
33 [A,b] = boundary_adjust(A, b, Pb, B_f_edges, @func_boundary, @func_c);% 调用调整器
34
35 %% 求解并进行误差分析

```

```

36 uh = A \ b;
37 err = [];
38 if fig_flag == true
39     show_eval(@func_real_u, uh, Pb, "Linear 1D FEM")% 调用可视化
40 end
41 if err_flag == true
42     err = err_eval(@func_real_u, left, right, uh, Nm, Nb, P, T, Pb, Tb, dim, type, hx, err_types);
43 end
44 end

```

3.7 1 维 FEM 程序依赖图

整个 FEM 程序就像精心设计的机构:

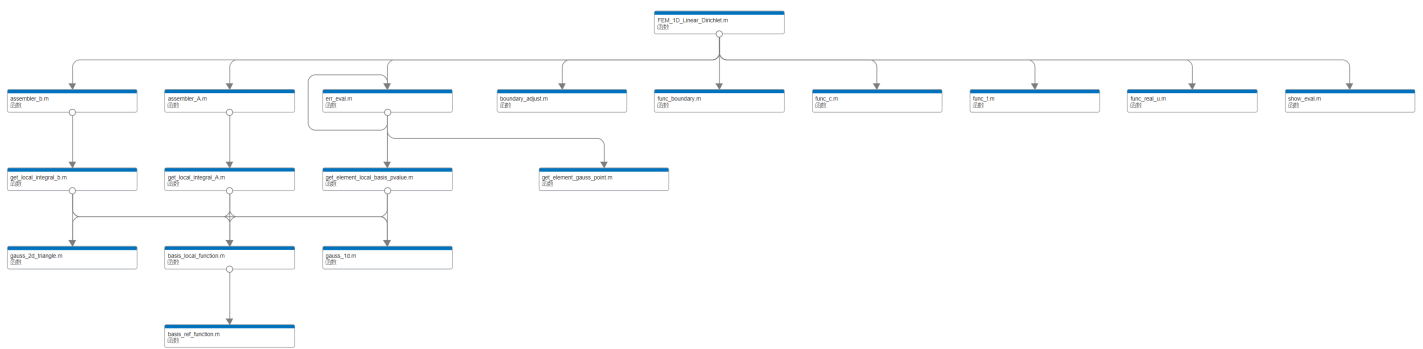


图 12: 程序依赖图

首先,先造好高斯积分和基函数这些“基础零件”,之后,搭建矩阵组装的生产线,最后配上误差分析和可视化这些辅助设备.现在你已经有了了一套完整的 1 维 **FEM** 工具包,接下来可以尝试换更复杂的零件(比如高阶元),或者造更大的模型(二维问题)了.

4 2 维 FEM 程序

4.1 实现两点式直线方程

为了给后面的二维线积分做铺垫, 先来实现一个简单的两点式直线方程: 给定平面内任意两点 $P_1(x_1, y_1), P_2(x_2, y_2)$, 还有评估点 x , 函数将会输出直线在该点处的 y 值.

Listing 21: line_func

```
1 % Get a line function from given two points
2 % P_1 = [x1 y1]; P_2 = [x2 y2];
3 function result = line_func(x, P_1, P_2)
4 if P_1(1) == P_2(1)
5     result = P_1(1);
6 elseif P_1(2) == P_2(2)
7     result = P_2(2);
8 else
9     result = (P_1(2) - P_2(2)) / (P_1(1) - P_2(1)) * (x - P_1(1)) + P_1(2);
```


4.2 实现二维线积分

所谓二维线积分, 实际上是指第一型曲线积分和第二型曲线积分的统称: 一个是对标量函数积分, 一个是对向量函数积分. 在这里, 我们希望求解的线积分为:

$$\int_{e_k} f(x, y) ds.$$

这里 e_k 是一条直线, 而 $f(x, y)$ 是一个标量函数, 所以这个线积分是第一型曲线积分.

为了使用高斯积分, 我们先将 e_k 参数化:

$$e_k : \begin{cases} x = x(t), \\ y = y(t). \end{cases}$$

设起始点 t_0 , 结束点 t_1 , 由第一型曲线积分的计算法则有:

$$\int_{e_k} f(x, y) ds = \int_{t_0}^{t_1} f(x(t), y(t)) \sqrt{x'^2(t) + y'^2(t)} dt.$$

而 e_k 是一条直线, 当 $e_k = kx + b$ 时, 设起始点 x_0 , 结束点 x_1 , 有:

$$e_k : \begin{cases} x = x, \\ y = kx + b. \end{cases} \int_{e_k} f(x, y) ds = \int_{x_0}^{x_1} f(x, kx + b) \sqrt{1 + k^2} dx.$$

当 e_k 平行于 y 轴时, 设起始点 y_0 , 结束点 y_1 , 有:

$$e_k : \begin{cases} x = x_0, \\ y = y. \end{cases} \int_{e_k} f(x, y) ds = \int_{y_0}^{y_1} f(x_0, y) \sqrt{0 + 1} dy = \int_{y_0}^{y_1} f(x_0, y) dy.$$

将上述所有可能总结如下:

表 1: 各直线形态总结

e_k 形态	积分形式	如何实现
$x = x_0, y_0 \leq y \leq y_1$ 型	$\int_{y_0}^{y_1} f(x_0, y) dy.$	固定 x , 取 $[y_0, y_1]$ 上的高斯点与权重.
$y = kx + b, x_0 \leq x \leq x_1$ 型	$\int_{x_0}^{x_1} f(x, kx + b) \sqrt{1 + k^2} dx.$	x 取 $[x_0, x_1]$ 上的高斯点与权重, y 将 x 取得的高斯点做变换 $kx + b$.
$y = y_0, x_0 \leq x \leq x_1$ 型	$\int_{x_0}^{x_1} f(x, y_0) dx.$	$k = 0$ 的 $kx + b$ 型.

注意, 在 PPT 中并没有明确指出 $\sqrt{1 + k^2}$, 但是在实际计算时, 这是必需的:

- Case 1: If a boundary edge is vertical, then it can be described as $x = c$ ($y_1 \leq y \leq y_2$). The y -coordinates of the Gauss quadrature nodes on this boundary edge and the Gauss quadrature weights can be obtained from the 1D local Gauss quadrature on $[y_1, y_2]$. And the x -coordinates of the Gauss quadrature nodes are fixed to be c .
- Case 2: If a boundary edge is horizontal, then it can be described as $y = c$ ($x_1 \leq x \leq x_2$). The x -coordinates of the Gauss quadrature nodes on this boundary edge and the Gauss quadrature weights can be obtained from the 1D local Gauss quadrature on $[x_1, x_2]$. And the y -coordinates of the Gauss quadrature nodes are fixed to be c .
- Case 3: Otherwise, a boundary edge can be described as $y = ax + b$ ($x_1 \leq x \leq x_2$). The x -coordinates of the Gauss quadrature nodes on this boundary edge and the Gauss quadrature weights can be obtained from the 1D local Gauss quadrature on $[x_1, x_2]$. And the y -coordinates of the Gauss quadrature nodes are obtained from $y = ax + b$.
- The case 3 with $a = 0$ and $b = c$ is equivalent to case 2. Hence case 2 and case 3 can be combined into one case.

图 13: 注意需要乘以 $\sqrt{1 + k^2}$

Listing 22: gauss_2d_line

```

1 % 高斯-勒让德求积(optional) 2D 第一型曲线积分
2 % f is a function handle which can feed in (x,y)
3 function result = gauss_2d_line(f,a,b,P_1,P_2,type)
4 gauss_points = [0,-0.3242534234,-0.6133714327,-0.8360311073,-0.9681602395,0.3242534234,0.6133714327,
    0.8360311073, 0.9681602395];
5 gauss_weights = [0.330239355,0.312347077,0.2606106964,...
6 0.1806481607,0.08127438836,0.312347077,0.2606106964,0.1806481607,0.08127438836];
7 if type == 'v'% 积分线垂直于x轴,此时a b为[ymin ymax],line_func为积分曲线@(x)c0
8     gauss_mapped_pointed = (b - a) .* gauss_points ./ 2 + (a + b) ./ 2;
9     result = (b - a)/2 * gauss_weights * f(line_func(gauss_mapped_pointed, P_1, P_2),
    gauss_mapped_pointed)';
10 else% 其他情况,此时a b为[xmin xmax],line_func为积分曲线@(x)ax+b
11     gauss_mapped_pointed = (b - a) .* gauss_points ./ 2 + (a + b) ./ 2;
12     k = (P_1(2) - P_2(2)) / (P_1(1) - P_2(1));
13     result = sqrt(1 + k^2) * (b - a)/2 * gauss_weights * f(gauss_mapped_pointed,line_func(
    gauss_mapped_pointed, P_1, P_2))';
14 end

```

4.3 实现三角单元上的 Gauss-Legendre 求积公式

设有标准三角元 T , 在 T 上的高斯积分点为 (ξ_k, η_k) , 积分系数为 w_k , 则有:

$$\iint_T f(\xi, \eta) d\xi d\eta \approx \sum_{k=0}^n w_k f(\xi_k, \eta_k).$$

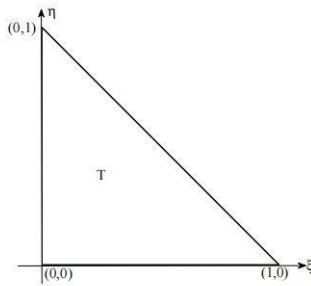


图 14: 标准三角元

下面, 我们借助形函数将标准三角元 T 变换到任意的三角元 K 上:

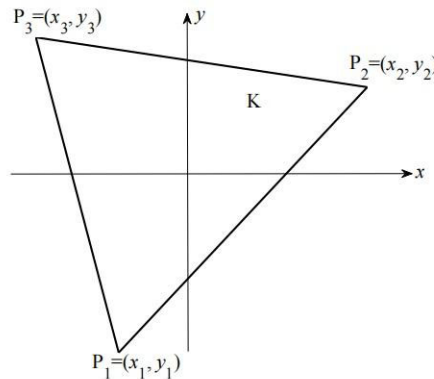


图 15: 任意三角元

对应的变换为:

$$\begin{cases} N_1(\xi, \eta) = 1 - \xi - \eta, \\ N_2(\xi, \eta) = \xi, \\ N_3(\xi, \eta) = \eta, \end{cases}$$

$$x = P(\xi, \eta) = x_1 N_1(\xi, \eta) + x_2 N_2(\xi, \eta) + x_3 N_3(\xi, \eta),$$

$$y = Q(\xi, \eta) = y_1 N_1(\xi, \eta) + y_2 N_2(\xi, \eta) + y_3 N_3(\xi, \eta).$$

因此, 在 K 上的高斯求积公式为:

$$\iint_K f(x, y) dx dy = \iint_T f(P(\xi, \eta), Q(\xi, \eta)) |J(\xi, \eta)| d\xi d\eta \approx \sum_{k=0}^n w_k f(P(\xi_k, \eta_k), Q(\xi_k, \eta_k)) |J(\xi_k, \eta_k)|.$$

对应三角元高斯点与高斯权重可自行查阅相关资料, 下面直接给出对应程序:

Listing 23: gauss_2d_triangle

```

1 % func need to be a function handle which can feed in (x,y).
2 % vertices is a form like [x1 x2 x3...;y1 y2 y3...]
3 function result = gauss_2d_triangle(func,vertices)
4 gauss_reference_weights = [64/81*(1-0)/8,100/324*(1-sqrt(3/5))/8,100/324*(1-sqrt(3/5))/8,100/324*(1+
    sqrt(3/5))/8,100/324*(1+sqrt(3/5))/8,40/81*(1-0)/8,40/81*(1-0)/8,40/81*(1-sqrt(3/5))/8,40/81*(1+
    sqrt(3/5))/8];
5 gauss_reference_points = [1/2,(1+sqrt(3/5))/2,(1+sqrt(3/5))/2,(1-sqrt(3/5))/2,(1-sqrt(3/5))/2, 1/2,
    1/2, (1+sqrt(3/5))/2,(1-sqrt(3/5))/2;...
6                             1/4,(1-sqrt(3/5))*(1+sqrt(3/5))/4,(1-sqrt(3/5))*(1-sqrt(3/5))/4,(1+sqrt
    (3/5))*(1+sqrt(3/5))/4,(1+sqrt(3/5))*(1-sqrt(3/5))/4,(1+sqrt(3/5))/4,(1-sqrt(3/5))/4,(1-sqrt
    (3/5))/4,(1+sqrt(3/5))/4];
7 det_J = det([vertices(:,2) - vertices(:,1),vertices(:,3) - vertices(:,1)]);
8 gauss_points_x = vertices(1,1) .* shape_func(gauss_reference_points(1,:),gauss_reference_points(2,:),
    ,1)...
9               + vertices(1,2) .* shape_func(gauss_reference_points(1,:),gauss_reference_points(2,:),
    ,2)...
10              + vertices(1,3) .* shape_func(gauss_reference_points(1,:),gauss_reference_points(2,:),
    ,3);
11 gauss_points_y = vertices(2,1) .* shape_func(gauss_reference_points(1,:),gauss_reference_points(2,:),
    ,1)...
12              + vertices(2,2) .* shape_func(gauss_reference_points(1,:),gauss_reference_points(2,:),
    ,2)...
13              + vertices(2,3) .* shape_func(gauss_reference_points(1,:),gauss_reference_points(2,:),
    ,3);
14 result = sum(gauss_reference_weights .* det_J .* func(gauss_points_x,gauss_points_y),"all");
15 end
16
17 function result = shape_func(x,y,type)
18 if type == 1
19     result = 1 - x - y;
20 elseif type == 2
21     result = x;
22 elseif type == 3
23     result = y;
24 end
25 end

```

4.4 实现信息矩阵整合函数

当维数来到 2 维时, 网格世界的索引管理变得格外热闹, 这时候为了方便之后的使用, 我们来编写一个输出各信息矩阵的子函数, 下面的程序以线性元和二单元为例.

4.4.1 了解坐标-索引关系

要想高效的转换各信息矩阵, 我们需要了解各个坐标-索引的关系, 首先, 我们展示一个全局的示意图, 随后一步步说明如何转换坐标-索引.

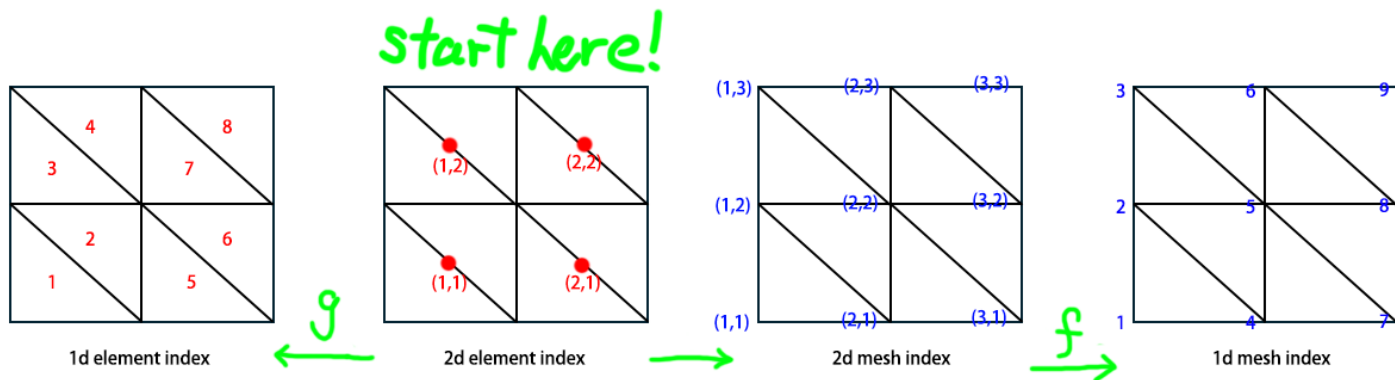


图 16: 坐标-索引转换图

在二维网格系统中, 需要建立四类关键的坐标-索引的对应关系:

1. **单元全局编号 (1D 索引)**: 为每个网格单元分配唯一全局编号
2. **单元自然坐标 (2D 索引)**: 采用 (x, y) 形式表示每个网格单元在网格中的逻辑位置
3. **网格节点自然坐标 (2D 索引)**: 每个网格顶点的逻辑坐标表示
4. **网格节点全局编号 (1D 索引)**: 为每个网格顶点分配唯一全局编号

4.4.2 实现 T 阵

我们先来完成 T 阵的生成. 首先, 我们利用 x, y 方向上的网格个数 N_x, N_y 生成单元的 2D 索引:

```
1 [m_idx_X, m_idx_Y] = meshgrid(1:1:Nx, 1:1:Ny); % 网格单元自然坐标
```

随后, 我们要构造一个多值函数 g , 它把单元的 2 维索引映到单元的 1 维索引:

$$g(x, y) = \begin{cases} 2N_y(x-1) + 2y, & \text{上半部分的 1D 索引,} \\ 2N_y(x-1) + 2y - 1, & \text{下半部分的 1D 索引.} \end{cases}$$

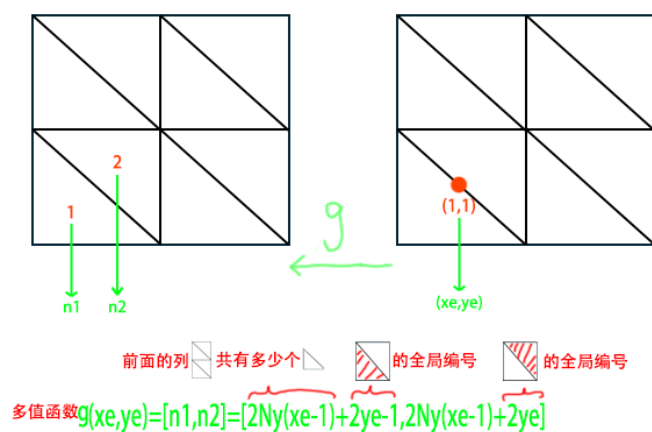


图 17: 多值函数 g

我们把所有单元的 2D 索引喂给 g , 就获得了所有单元的全局编号, 而对于从单元的 2 维索引转换到网格的 2 维索引, 实际上将其看做是左下部分, 然后简单对坐标 ± 1 即可——因为每个网格节点都可以确定它们旁边的小伙伴:

$$\begin{bmatrix} (x, y+1) & \leftrightarrow & (x+1, y+1) \\ \updownarrow & \nearrow & \updownarrow \\ (x, y) & \leftrightarrow & (x+1, y) \end{bmatrix}.$$

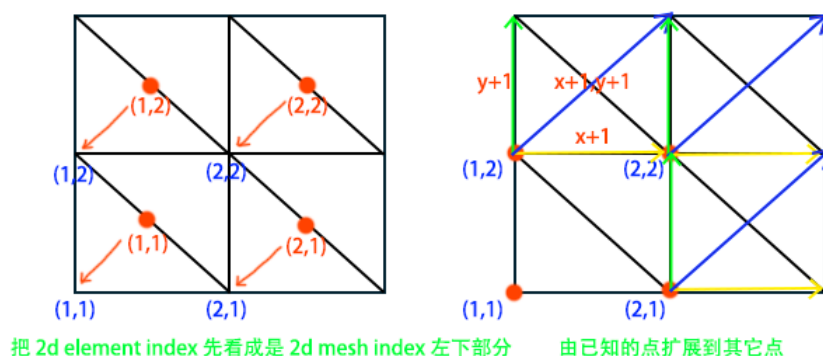


图 18: 点的扩展

最后我们依靠 f 来获得网格的 1 维索引:

$$f(x, y) = (N_y + 1)(x - 1) + y.$$

别忘了我们 T 阵的行是以局部索引为规则存储的, 而我们的局部索引为:

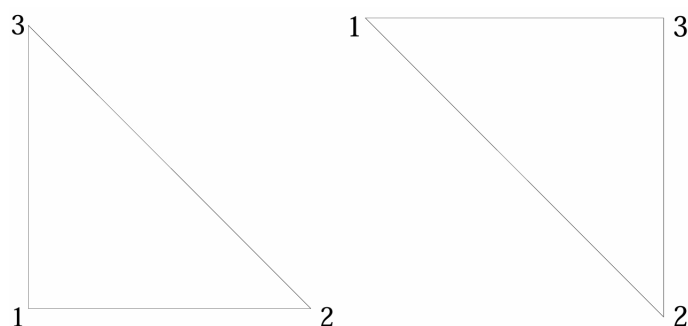


图 19: 三角元局部索引

将这一套思路写成代码就是:

Listing 24: `m_index_to_tri_T`

```
1 % Convert natural elements index to matrix T
```

```

2 function T = m_index_to_tri_T(flatten_eX,flatten_eY,Ny)
3 f = @(x,y) (Ny + 1) * (x - 1) + y;% Convert natural mesh nodes index to 1D index
4 to_T = @(x,y) [(x - 1) * 2 * Ny + 2 * y - 1,(x - 1) * 2 * Ny + 2 * y;...% 单元的全局索引
5               f(x,y),f(x,y + 1);...% 左下,左上
6               f(x + 1,y),f(x + 1,y);...% 右下,右下
7               f(x,y + 1),f(x + 1,y + 1)];% 左上,右上
8 % natural elements index -> 1D elements index & natural mesh nodes index -> 1D mesh nodes index
9 T = to_T(flatten_eX,flatten_eY);
10 T(:,T(1,:)) = T(:,sort(T(1,:)));% 按照自然顺序
11 T = T(2:end,:);% 抹去第一列
12 end

```

具体来说,我们首先将单元的坐标展平成向量以便进行向量化处理:

$$\begin{aligned} \text{flatten_eX} &= \begin{pmatrix} 1 & 1 & 2 & 2 \end{pmatrix}, \\ \text{flatten_eY} &= \begin{pmatrix} 1 & 2 & 1 & 2 \end{pmatrix}. \end{aligned}$$

利用 `to_T` 函数一次性处理所有坐标,从而得出原始的 T 矩阵(第一行是全局单元编号,剩余行是网格点的局部编号),可以看出,这个 T 还没有按照自然顺序排列全局单元编号:

$$T = \begin{pmatrix} 1 & 3 & 5 & 7 & 2 & 4 & 6 & 8 \\ 1 & 2 & 4 & 5 & 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 & 4 & 5 & 7 & 8 \\ 2 & 3 & 5 & 6 & 5 & 6 & 8 & 9 \end{pmatrix}.$$

先依据第一行进行排序:

$$T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 2 & 3 & 4 & 5 & 5 & 6 \\ 4 & 4 & 5 & 5 & 7 & 7 & 8 & 8 \\ 2 & 5 & 3 & 6 & 5 & 8 & 6 & 9 \end{pmatrix}.$$

再删去第一行,得到最终的 T 矩阵:

$$T = \begin{pmatrix} 1 & 2 & 2 & 3 & 4 & 5 & 5 & 6 \\ 4 & 4 & 5 & 5 & 7 & 7 & 8 & 8 \\ 2 & 5 & 3 & 6 & 5 & 8 & 6 & 9 \end{pmatrix}.$$

4.4.3 实现 Tb 阵

对于线性元来说, $T = Tb$.但是由于二次元的有限元节点与网格节点不同,所以我们需要另一套坐标(有限元节点坐标):

start here!

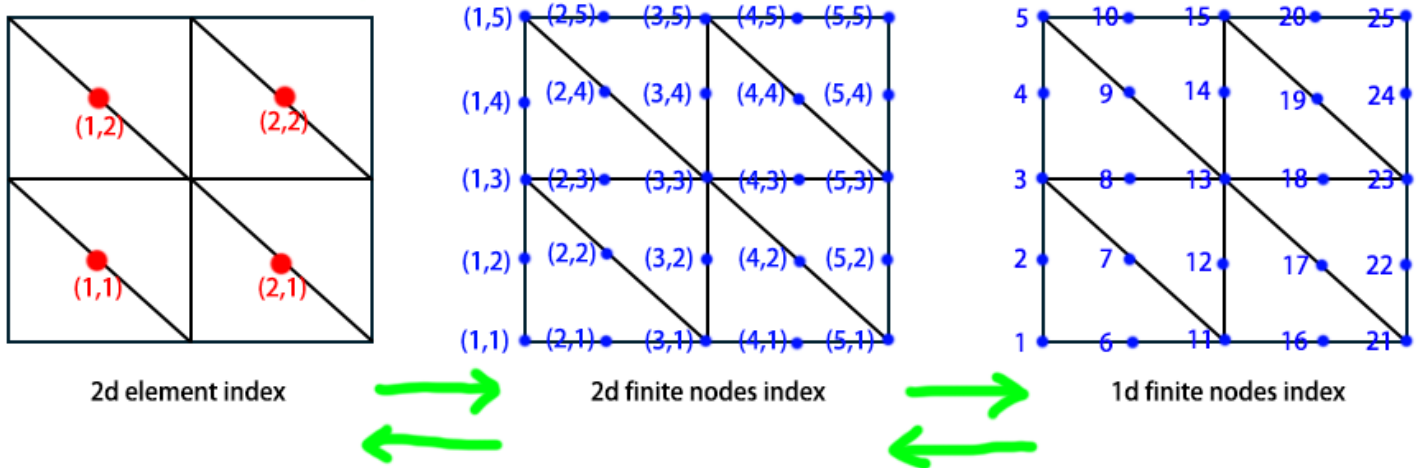


图 20: 有限元节点坐标

我们以“不变”应万变,所作的唯一点调整就是在将单元的二维坐标转换到有限元的二维坐标时:

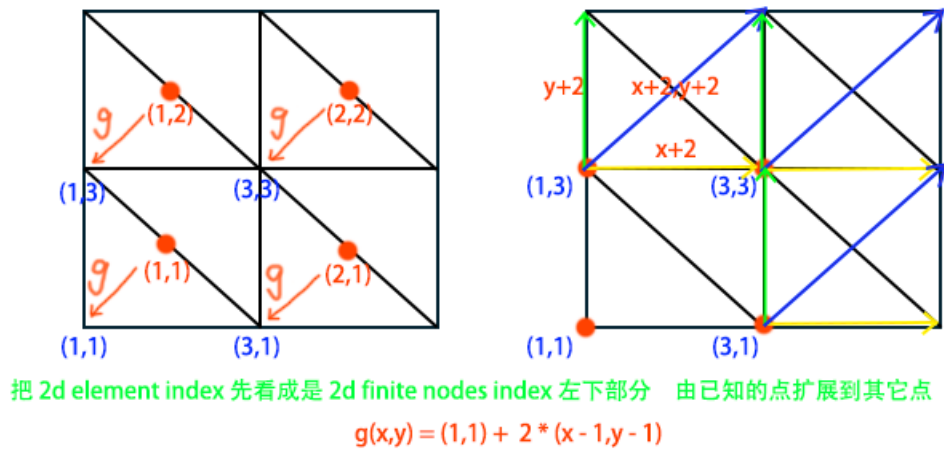


图 21: 有限元节点坐标转换

在动手敲代码之前,确认一下 Tb 阵的局部索引:

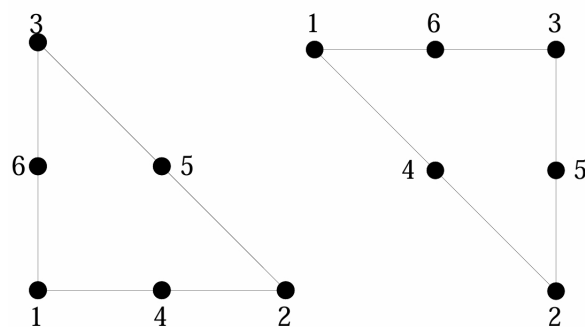


图 22: 三角元有限元局部索引

写成代码就是:

Listing 25: f_index_to_qua_Tb

```
1 % Convert natural finite nodes index to matrix Tb (Quadratic Only)
2 function Tb = f_index_to_qua_Tb(flatten_eX,flatten_eY,Nfy,Ny)
3 f = @(x,y) Nfy * (x - 1) + y;% Convert natural finite nodes index to 1D index
```



```

4 g = @(x) 1 + 2 * (x - 1); % Convert natural mesh x nodes index to local finite x index (左下角),y的一
   样,因此省略
5 to_T = @(x,y) [(x - 1) * 2 * Ny + 2 * y - 1,(x - 1) * 2 * Ny + 2 * y;...% 单元的全局索引
6             f(g(x),g(y)),f(g(x),g(y) + 2);...% 左下,左上
7             f(g(x) + 2,g(y)),f(g(x) + 2,g(y));...% 右下,右下
8             f(g(x),g(y) + 2),f(g(x) + 2,g(y) + 2);...% 左上,右上
9             f(g(x) + 1,g(y)),f(g(x) + 1,g(y) + 1);...% 中下,正中
10            f(g(x) + 1,g(y) + 1),f(g(x) + 2,g(y) + 1);...% 正中,右中
11            f(g(x),g(y) + 1),f(g(x) + 1,g(y) + 2)]; % 左中,中上
12 % natural elements indeg(x) -> 1D elements indeg(x) & natural finite nodes indeg(x) -> 1D finite
   nodes indeg(x)
13 Tb = to_T(flatten_eX,flatten_eY);
14 Tb(:,Tb(1,:)) = Tb(:,sort(Tb(1,:))); % 按照自然顺序
15 Tb = Tb(2:end,:); % 抹去第一列
16 end

```

4.4.4 实现网格边界信息矩阵

这里按照逆时针的顺序,从左下角开始依次取网格的 1 维索引即可,利用之前我们的坐标关系有:

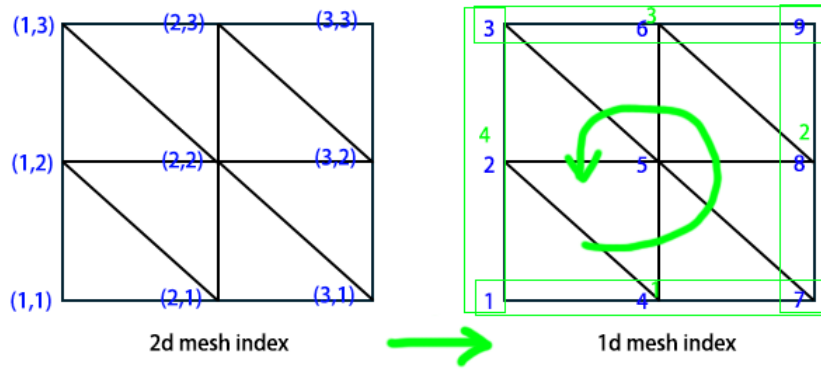


图 23: 网格边界坐标转换

Listing 26: edges_to_tri_midx

```

1 % Convert T to boundary edges mesh nodes with boundary type
2 % Local index: down -> right -> up -> left
3 function B = edges_to_tri_midx(e_idx_X,e_idx_Y,T,Nx,Ny,boundary_type)
4 to_idx_l = @(x,y)(x - 1) * 2 * Ny + 2 * y - 1; % Convert natural elements index to 1D index (left)
5 to_idx_r = @(x,y)(x - 1) * 2 * Ny + 2 * y; % Convert natural elements index to 1D index (right)
6 down_m_idx = to_idx_l(reshape(e_idx_X(1,:),1,[],reshape(e_idx_Y(1,:),1,[])));
7 right_m_idx = to_idx_r(reshape(e_idx_X(:,end),1,[],reshape(e_idx_Y(:,end),1,[])));
8 up_m_idx = flip(to_idx_r(reshape(e_idx_X(end,:),1,[],reshape(e_idx_Y(end,:),1,[])));
9 left_m_idx = flip(to_idx_l(reshape(e_idx_X(:,1),1,[],reshape(e_idx_Y(:,1),1,[])));
10 temp = [down_m_idx,right_m_idx,up_m_idx,left_m_idx;...
11         T([1 2],down_m_idx),T([2 3],right_m_idx),T([3 1],up_m_idx),T([3 1],left_m_idx)]; % find
   corresponding mesh nodes 1d index
12 B = [repelem(boundary_type,[Nx,Ny,Nx,Ny]);temp];
13 end

```

这里的 flip 是翻转数组的函数,因为我们要依照逆时针编写矩阵,最后的 repelem 实际上就是将边界类型分配到第一行。

4.4.5 实现有限元边界信息矩阵

同理实现有限元边界信息矩阵,唯一的区别就是将网格索引换成了有限元索引,当然对于线性元来说,有限元索引就是网格索引,但是二次元就不一样了:

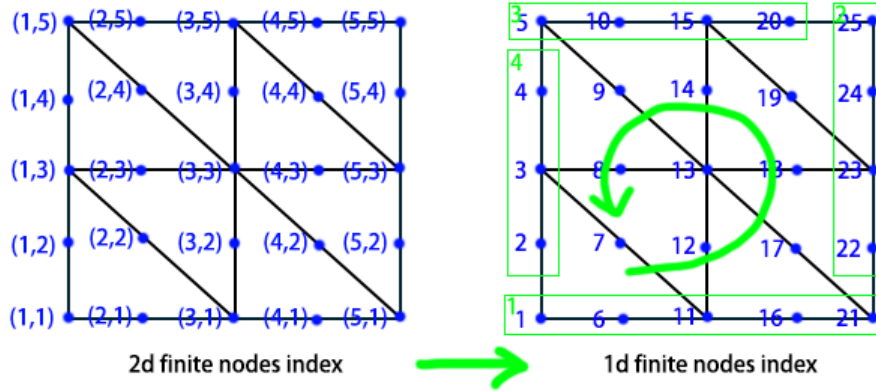


图 24: 有限元边界坐标转换

Listing 27: edges_to_tri_fidx

```
1 % Convert finite nodes natural index to boundary 1D finite nodes index with boundary type
2 % Local index: down -> right -> up -> left
3 function B = edges_to_tri_fidx(f_idx_X,f_idx_Y,Nfx,Nfy,boundary_type)
4 to_idx = @(x,y)(x - 1) * Nfy + y;% Convert finite nodes natural index to 1D index
5 down_f_idx = to_idx(reshape(f_idx_X(1,:),1,[]),reshape(f_idx_Y(1,:),1,[]));
6 right_f_idx = to_idx(reshape(f_idx_X(2:end,end),1,[]),reshape(f_idx_Y(2:end,end),1,[]));
7 up_f_idx = flip(to_idx(reshape(f_idx_X(end,1:end - 1),1,[]),reshape(f_idx_Y(end,1:end - 1),1,[])));
8 left_f_idx = flip(to_idx(reshape(f_idx_X(2:end - 1,1),1,[]),reshape(f_idx_Y(2:end - 1,1),1,[])));
9 temp = [down_f_idx,right_f_idx,up_f_idx,left_f_idx];
10 B = [repelem(boundary_type,[Nfx,Nfy - 1,Nfx - 1,Nfy - 2]);temp];
11 end
```

4.4.6 确保有限元边界信息矩阵在交点处的正确性

结束了么——先别高兴的太早了. 我们要确保当同时使用 3 种边界条件时,Dirichlet 与 Neumann 和 Robin 的交界点处的边界条件是 Dirichlet 的,这一步通过网格边界信息矩阵来对有限元边界信息矩阵做调整——对交界点无脑复制粘贴 if 语句吧:

Listing 28: check_legal

```
1 % make sure apply Dirichlet condition in intersection.
2 function B_f_edges = check_legal(B_m_edges,B_f_edges,Nx,Ny,Nfx,Nfy)
3 % Neumann
4 if (B_m_edges(1,1) == -2 && B_m_edges(1,end) == -1) || (B_m_edges(1,1) == -1 && B_m_edges(1,end) == -2)
5     B_f_edges(1,1) = -1;
6 end
7 if (B_m_edges(1,Nx) == -2 && B_m_edges(1,Nx + 1) == -1) || (B_m_edges(1,Nx) == -1 && B_m_edges(1,Nx + 1) == -2)
8     B_f_edges(1,Nfx) = -1;
9 end
10 if (B_m_edges(1,Nx + Ny) == -2 && B_m_edges(1,Nx + Ny + 1) == -1) || (B_m_edges(1,Nx + Ny) == -1 && B_m_edges(1,Nx + Ny + 1) == -2)
```

```

11     B_f_edges(1,Nfx + Nfy - 1) = -1;
12 end
13 if (B_m_edges(1,2 * Nx + Ny) == -2 && B_m_edges(1,2 * Nx + Ny + 1) == -1) || (B_m_edges(1,2 * Nx +
    Ny) == -1 && B_m_edges(1,2 * Nx + Ny + 1) == -2)
14     B_f_edges(1,2 * Nfx + Nfy - 2) = -1;
15 end
16 % Robin
17 if (B_m_edges(1,1) == -3 && B_m_edges(1,end) == -1) || (B_m_edges(1,1) == -1 && B_m_edges(1,end) ==
    -3)
18     B_f_edges(1,1) = -1;
19 end
20 if (B_m_edges(1,Nx) == -3 && B_m_edges(1,Nx + 1) == -1) || (B_m_edges(1,Nx) == -1 && B_m_edges(1,Nx
    + 1) == -3)
21     B_f_edges(1,Nfx) = -1;
22 end
23 if (B_m_edges(1,Nx + Ny) == -3 && B_m_edges(1,Nx + Ny + 1) == -1) || (B_m_edges(1,Nx + Ny) == -1 &&
    B_m_edges(1,Nx + Ny + 1) == -3)
24     B_f_edges(1,Nfx + Nfy - 1) = -1;
25 end
26 if (B_m_edges(1,2 * Nx + Ny) == -3 && B_m_edges(1,2 * Nx + Ny + 1) == -1) || (B_m_edges(1,2 * Nx +
    Ny) == -1 && B_m_edges(1,2 * Nx + Ny + 1) == -3)
27     B_f_edges(1,2 * Nfx + Nfy - 2) = -1;
28 end
29 end

```

4.4.7 实现信息矩阵整合函数

最后让我们把所有“拼图”拼接好:

Listing 29: get_all_info

```

1 function [Nf,Nm,Nb,P,T,Pb,Tb,B_m_edges,B_f_edges,f_X,f_Y] = get_all_info(left,right,down,up,hx,hy,
    boundary_type,dim,type)
2 %% 获取网格信息
3 Nx = (right - left) / hx;% x网格单元数
4 Ny = (up - down) / hy;% y网格单元数
5
6 %% 2D_Linear
7 if dim == 2 && type == 1
8 % 获取有限元信息
9 Nfx = Nx + 1;% x有限元节点数
10 Nfy = Ny + 1;% y有限元节点数
11 Nf = Nfx * Nfy;% 有限元节点数
12 Nm = 2 * Nx * Ny;% 网格单元数
13 Nb = 3;% 局部基函数个数
14
15 % 初始化其余信息
16 [m_idx_X,m_idx_Y] = meshgrid(1:1:Nx,1:1:Ny);% 网格单元自然坐标
17 [m_X,m_Y] = meshgrid(left:hx:right,down:hy:up);% 网格坐标
18 [f_idx_X,f_idx_Y] = meshgrid(1:1:Nfx,1:1:Nfy);% 有限元节点自然坐标
19 [f_X,f_Y] = deal(m_X,m_Y);% 有限元坐标
20

```

```

21 % 构建P,T,Pb,Tb信息矩阵
22 P = [reshape(m_X,1,[]);reshape(m_Y,1,[])];% 网格点坐标矩阵
23 T = m_index_to_tri_T(reshape(m_idx_X,1,[]),reshape(m_idx_Y,1,[]),Ny);% 网格点索引矩阵
24 Pb = P;% 有限元点坐标矩阵
25 Tb = T;% 有限元点索引矩阵
26 B_m_edges = edges_to_tri_mbidx(m_idx_X,m_idx_Y,T,Nx,Ny,boundary_type);% 网格边界信息矩阵(第一行是边界类型,第二行是网格边索引,其余行是网格边端点)
27 B_f_edges = edges_to_tri_fbidx(f_idx_X,f_idx_Y,Nfx,Nfy,boundary_type);% 有限元边界信息矩阵(第一行是边界类型,第二行是有限元边节点)
28 B_f_edges = check_legal(B_m_edges,B_f_edges,Nx,Ny,Nfx,Nfy);% 确保正确施加边界条件
29 end
30
31 %% 2D_Quadratic
32 if dim == 2 && type == 2
33 % 获取有限元信息
34 Nfx = 2 * Nx + 1;% x有限元节点数
35 Nfy = 2 * Ny + 1;% y有限元节点数
36 Nf = Nfx * Nfy;% 有限元节点数
37 Nm = 2 * Nx * Ny;% 网格单元数
38 Nb = 6;% 局部基函数个数
39
40 % 初始化其余信息
41 [m_idx_X,m_idx_Y] = meshgrid(1:1:Nx,1:1:Ny);% 网格单元自然坐标
42 [m_X,m_Y] = meshgrid(left:hx:right,down:hy:up);% 网格坐标
43 [f_idx_X,f_idx_Y] = meshgrid(1:1:Nfx,1:1:Nfy);% 有限元节点自然坐标
44 [f_X,f_Y] = meshgrid(left:hx/2:right,down:hy/2:up);% 有限元坐标
45
46 % 构建P,T,Pb,Tb信息矩阵
47 P = [reshape(m_X,1,[]);reshape(m_Y,1,[])];% 网格点坐标矩阵
48 T = m_index_to_tri_T(reshape(m_idx_X,1,[]),reshape(m_idx_Y,1,[]),Ny);% 网格点索引矩阵
49 Pb = [reshape(f_X,1,[]);reshape(f_Y,1,[])];% 有限元点坐标矩阵
50 Tb = f_index_to_qua_Tb(reshape(m_idx_X,1,[]),reshape(m_idx_Y,1,[]),Nfy,Ny);% 有限元点索引矩阵
51 B_m_edges = edges_to_tri_mbidx(m_idx_X,m_idx_Y,T,Nx,Ny,boundary_type);% 网格边界信息矩阵(第一行是边界类型,第二行是网格边索引,其余行是网格边端点)
52 B_f_edges = edges_to_tri_fbidx(f_idx_X,f_idx_Y,Nfx,Nfy,boundary_type);% 有限元边界信息矩阵(第一行是边界类型,第二行是有限元边节点)
53 B_f_edges = check_legal(B_m_edges,B_f_edges,Nx,Ny,Nfx,Nfy);% 确保正确施加边界条件
54 end

```

4.5 更新参考元上的基函数

因为之前我们是使用 1 维的基函数,所以现在我们需要补充 2 维的基函数到对应的程序里,这里以线性元为例,其它基函数根据需要自行补充:

```

1 %% dim = 2 type = 1 2D_Linear FE (triangle)
2 if dim == 2
3     if type == 1
4         if index == 1
5             if dx == 0 && dy == 0
6                 result = 1 - x - y;
7             elseif dx == 1 && dy == 0
8                 result = - 1;

```

```

9         elseif dx == 0 && dy == 1
10             result = - 1;
11         else
12             result = 0;
13         end
14     end
15     if index == 2
16         if dx == 0 && dy == 0
17             result = x;
18         elseif dx == 1 && dy == 0
19             result = 1;
20         else
21             result = 0;
22         end
23     end
24     if index == 3
25         if dx == 0 && dy == 0
26             result = y;
27         elseif dx == 0 && dy == 1
28             result = 1;
29         else
30             result = 0;
31         end
32     end
33 end
34 end

```

4.6 更新单元上的基函数

同样的,我们也需要更新单元上的基函数,这里以线性元和二单元为例,它们的计算方法都是:

$$J_n = \begin{pmatrix} x_{n2} - x_{n1} & x_{n3} - x_{n1} \\ y_{n2} - y_{n1} & y_{n3} - y_{n1} \end{pmatrix},$$

$$\hat{x} = \frac{(y_{n3} - y_{n1})(x - x_{n1}) - (x_{n3} - x_{n1})(y - y_{n1})}{|J_n|},$$

$$\hat{y} = \frac{(y_{n1} - y_{n2})(x - x_{n1}) + (x_{n2} - x_{n1})(y - y_{n1})}{|J_n|},$$

$$\psi_{ni}(x, y) = \hat{\psi}_i(\hat{x}, \hat{y}), i = 1, 2, \dots$$

$$\frac{\partial \psi_{ni}}{\partial x} = \frac{\partial \hat{\psi}_i}{\partial \hat{x}} \frac{y_{n3} - y_{n1}}{|J_n|} + \frac{\partial \hat{\psi}_i}{\partial \hat{y}} \frac{y_{n1} - y_{n2}}{|J_n|},$$

$$\frac{\partial \psi_{ni}}{\partial y} = \frac{\partial \hat{\psi}_i}{\partial \hat{x}} \frac{x_{n1} - x_{n3}}{|J_n|} + \frac{\partial \hat{\psi}_i}{\partial \hat{y}} \frac{x_{n2} - x_{n1}}{|J_n|},$$

$$\frac{\partial^2 \psi_{ni}}{\partial x^2} = \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x}^2} \frac{(y_{n3} - y_{n1})^2}{|J_n|^2} + 2 \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x} \partial \hat{y}} \frac{(y_{n3} - y_{n1})(y_{n1} - y_{n2})}{|J_n|^2} + \frac{\partial^2 \hat{\psi}_i}{\partial \hat{y}^2} \frac{(y_{n1} - y_{n2})^2}{|J_n|^2},$$

$$\frac{\partial^2 \psi_{ni}}{\partial y^2} = \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x}^2} \frac{(x_{n1} - x_{n3})^2}{|J_n|^2} + 2 \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x} \partial \hat{y}} \frac{(x_{n1} - x_{n3})(x_{n2} - x_{n1})}{|J_n|^2} + \frac{\partial^2 \hat{\psi}_i}{\partial \hat{y}^2} \frac{(x_{n2} - x_{n1})^2}{|J_n|^2},$$

$$\begin{aligned} \frac{\partial^2 \psi_{ni}}{\partial x \partial y} &= \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x}^2} \frac{(x_{n1} - x_{n3})(y_{n3} - y_{n1})}{|J_n|^2} + \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x} \partial \hat{y}} \frac{(x_{n1} - x_{n3})(y_{n1} - y_{n2})}{|J_n|^2} \\ &+ \frac{\partial^2 \hat{\psi}_i}{\partial \hat{x} \partial \hat{y}} \frac{(x_{n2} - x_{n1})(y_{n3} - y_{n1})}{|J_n|^2} + \frac{\partial^2 \hat{\psi}_i}{\partial \hat{y}^2} \frac{(x_{n2} - x_{n1})(y_{n1} - y_{n2})}{|J_n|^2}. \end{aligned}$$

```

1 %% dim = 2 2D_Linear FE && 2D_Quadratic FE (triangle)
2 % vertices = [x1 x2 x3; y1 y2 y3]
3 if dim == 2
4     det_J = det([vertices(:,2) - vertices(:,1), vertices(:,3) - vertices(:,1)]);
5     x_hat = ((vertices(2,3) - vertices(2,1)) .* (x - vertices(1,1)) - (vertices(1,3) - vertices(1,1))
6     ) .* (y - vertices(2,1))) ./ det_J;
7     y_hat = ((vertices(1,2) - vertices(1,1)) .* (y - vertices(2,1)) - (vertices(2,2) - vertices(2,1))
8     ) .* (x - vertices(1,1))) ./ det_J;
9     x_13 = (vertices(1,1) - vertices(1,3)) ./ det_J;
10    x_21 = (vertices(1,2) - vertices(1,1)) ./ det_J;
11    y_31 = (vertices(2,3) - vertices(2,1)) ./ det_J;
12    y_12 = (vertices(2,1) - vertices(2,2)) ./ det_J;
13    if dx == 0 && dy == 0
14        result = basis_ref_function(x_hat, y_hat, dim, type, index, dx, dy);
15    elseif dx == 1 && dy == 0
16        result = basis_ref_function(x_hat, y_hat, dim, type, index, 1, 0) .* y_31...
17        + basis_ref_function(x_hat, y_hat, dim, type, index, 0, 1) .* y_12;
18    elseif dx == 0 && dy == 1
19        result = basis_ref_function(x_hat, y_hat, dim, type, index, 1, 0) .* x_13...
20        + basis_ref_function(x_hat, y_hat, dim, type, index, 0, 1) .* x_21;
21    elseif dx == 1 && dy == 1
22        result = basis_ref_function(x_hat, y_hat, dim, type, index, 2, 0) .* x_13 .* y_31...
23        + basis_ref_function(x_hat, y_hat, dim, type, index, 1, 1) .* x_13 .* y_12...
24        + basis_ref_function(x_hat, y_hat, dim, type, index, 1, 1) .* x_21 .* y_31...
25        + basis_ref_function(x_hat, y_hat, dim, type, index, 0, 2) .* x_21 .* y_12;
26    elseif dx == 2 && dy == 0
27        result = basis_ref_function(x_hat, y_hat, dim, type, index, 2, 0) .* y_31 .* y_31...
28        + 2 .* basis_ref_function(x_hat, y_hat, dim, type, index, 1, 1) .* y_31 .* y_12...
29        + basis_ref_function(x_hat, y_hat, dim, type, index, 0, 2) .* y_12 .* y_12;
30    elseif dx == 0 && dy == 2
31        result = basis_ref_function(x_hat, y_hat, dim, type, index, 2, 0) .* x_13 .* x_13...
32        + 2 .* basis_ref_function(x_hat, y_hat, dim, type, index, 1, 1) .* x_13 .* x_21...
33        + basis_ref_function(x_hat, y_hat, dim, type, index, 0, 2) .* x_21 .* x_21;
34    else
35        result = 0;%这里偷懒了
36    end
37 end

```

4.7 实现 2 维 FEM

终于, 可以将 1 维程序更新至 2 维了. 我们来完成最后的组装工作 (注意了: 对应的可视化函数和边界调整函数也需要依照自身需求更新, 计算各范数误差函数需要调整积分函数为 **gauss_2d_triangle**.)

Listing 30: FEM_2D_Linear_Dirichlet

```

1 %% Local Assembly FEM for 2D example
2 % 计算 - ·(c u) = -y(1-y)(1-x-x^2/2)exp(x+y)-x(1-x/2)(-3y-y^2)exp(x+y)
3 % c = 1;
4 % u = -1.5y(1-y)exp(-1+y) x = -1; u = 0.5y(1-y)exp(1+y) x = 1;
5 % u = -2x(1-x/2)exp(x-1) y = -1; u = 0 y = 1;
6 % real solution: xy(1-x/2)(1-y)exp(x+y)

```

```

7 % \omega = [-1,1]^2
8 % 输入: 网格单元数input_h [hx hy], 是否绘图fig_flag, 是否输出误差err_flag, 所需误差类型err_types(用""包裹的向量)
9 % 输出: 有限元节点处的解析解, 所选err_types的误差err
10 function [uh,err] = FEM_2D_Linear_Dirichlet(input_h,fig_flag,err_flag,err_types)
11 %% 输入网格信息
12 left = -1;
13 right = 1;
14 down = -1;
15 up = 1;
16 boundary_type = [-1 -1 -1 -1];% 边界条件 -1 -> 'D', -2 -> 'Neu', -3 -> 'Robin'
17
18 hx = input_h(1);% x 网格步长
19 hy = input_h(2);% y 网格步长
20
21 dim = 2;% 2D
22 type = 1;% Linear
23
24 [Nf,Nm,Nb,P,T,Pb,Tb,B_m_edges,B_f_edges,f_X,f_Y] = get_all_info(left,right,down,up,hx,hy,
    boundary_type,dim,type);
25
26 %% 组装刚度矩阵与荷载矩阵
27 A1 = assembler_A(2, Nf, Nf, Nm, P, T, Nb, Nb, [dim type], [dim type], Tb, Tb, @func_c, 1, 0, 1, 0);
28 A2 = assembler_A(2, Nf, Nf, Nm, P, T, Nb, Nb, [dim type], [dim type], Tb, Tb, @func_c, 0, 1, 0, 1);
29 b = assembler_b(2, Nf, Nm, P, T, Nb, [dim type], Tb, @func_f, 0, 0);
30 A = A1 + A2;
31 %% 使用边界条件信息矩阵调整边界条件
32 [A,b] = boundary_adjust(A,b,P,T,Pb,Tb,B_f_edges,B_m_edges,Nb,Nb,@func_boundary,@func_c,[],[],[],dim,
    type);% 调用调整器
33
34 %% 求解并进行误差分析
35 uh = A \ b;
36 err = [];
37 if fig_flag == true
38     show_eval(@func_real_u, uh, Pb, f_X, f_Y, "Linear 2D FEM")% 调用可视化
39 end
40 if err_flag == true
41     err = err_eval(@func_real_u, left, right, down, up, uh, Nm, Nb, P, T, Pb, Tb, dim, type, input_h
        , err_types);
42 end
43 end

```

4.8 2 维 FEM 程序依赖图

最后, 我们给出 2 维 FEM 程序的依赖图:

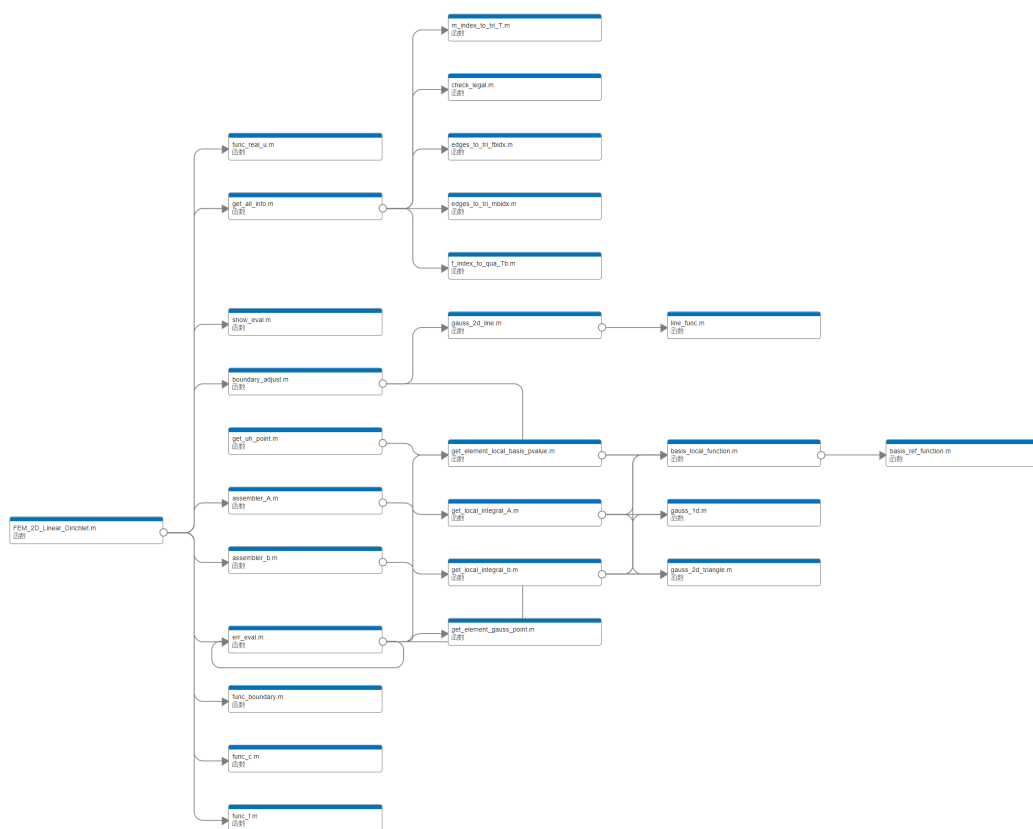


图 25: 2 维程序依赖图

恭喜你顺利完成新手教程! 不过, 真正的挑战才刚刚开始. 现在, 拿出你的 **FEM** 工具包, 运用你的智慧, 来击败你们自己的 **Boss** 吧! 别担心, 工具包里应有尽有, 你只需要像搭积木一样将它们巧妙地组合起来 (笑).