MARS2D 测试文档

1 LUD 分解

LUD 分解是用来求解系数矩阵为循环三对角矩阵的线性方程组的：

$$\boldsymbol{A}\boldsymbol{x}=\begin{bmatrix} b_1 & c_1 & & a_1 \\ a_2 & \ddots & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ c_n & & a_n & b_n \end{bmatrix}\boldsymbol{x}=\boldsymbol{b}. \tag{1}$$

在程序中主要用于循环三次样条插值中。

首先对 $\boldsymbol{A}$ 进行第一次分解：

$$\boldsymbol{A}=\begin{bmatrix} b_1 & c_1 & & a_1 \\ a_2 & \ddots & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ c_n & & a_n & b_n \end{bmatrix}=\begin{bmatrix} p_1 & & & \\ a_2 & p_2 & & \\ & \ddots & \ddots & \\ & & \ddots & \ddots \\ & & & a_n \quad p_n \end{bmatrix}\begin{bmatrix} 1 & q_1 & & & r_1 \\ & \ddots & \ddots & & \vdots \\ & & 1 & q_{n-2} & r_{n-2} \\ & & & 1 & r_{n-1} \\ s & & & & 1 \end{bmatrix}=\boldsymbol{L}\tilde{\boldsymbol{U}}, \tag{2}$$

其中

$$\begin{cases} p_1=b_1, q_1=c_1/p_1, r_1=a_1/p_1; \\ p_i=b_i-a_iq_{i-1}, q_i=c_i/p_i, r_i=-a_ir_{i-1}/p_i, i=2,\cdots,n-2; \\ p_{n-1}=b_{n-1}-a_{n-1}q_{n-2}, r_{n-1}=(c_{n-1}-a_{n-1}r_{n-2})/p_{n-1}; \\ p_n=b_n-a_nr_{n-1}, s=c_n/p_n. \end{cases} \tag{3}$$

再对 $\tilde{\boldsymbol{U}}$ 进行分解：

$$\tilde{\boldsymbol{U}}=\begin{bmatrix} 1 & q_1 & & & r_1 \\ & \ddots & \ddots & & \vdots \\ & & 1 & q_{n-2} & r_{n-2} \\ & & & 1 & r_{n-1} \\ s & & & & 1 \end{bmatrix}=\begin{bmatrix} 1 & q_1 & & & \\ & \ddots & \ddots & & \\ & & 1 & q_{n-2} & \\ & & & 1 & 0 \\ & & & & 1 \end{bmatrix}\begin{bmatrix} 1 & & & & t_1 \\ & \ddots & & & \vdots \\ & & 1 & & t_{n-2} \\ & & & 1 & t_{n-1} \\ s & & & & 1 \end{bmatrix}=\boldsymbol{U}\boldsymbol{D}, \tag{4}$$

其中

$$\begin{cases} t_{n-1}=r_{n-1}; \\ t_i=r_i-q_it_{i+1}, i=n-2,n-3,\cdots,1. \end{cases} \tag{5}$$

之后分别求解 $\boldsymbol{Lz}=\boldsymbol{b}$，$\boldsymbol{Uy}=\boldsymbol{z}$，$\boldsymbol{Dx}=\boldsymbol{y}$ 即可，这几个线性方程组都很容易求解。整体时间复杂度为 O(n)，n 为系数矩阵的阶数。

2 一种加减点方法：IMV

这里介绍一下谭焱师兄提出来的一种加减点方法。

`List` 在我们的程序中的缺点表现在建立临时变量慢，优点在于加减点是 O(1) 时间复杂度的，所以我们要进行改进就要找到一种方法，既能避免建立 `List` 临时变量，又同时保持加减点的时间复杂度是 O(1) 的。由于我们是基于 `Vector` 实现的，所以这种方法我们暂且叫它 IMV(Improved Vector) 方法。

2.1 容器

使用 `Vector` 来存储示踪点列，这样在每一个时间步中就是创建一个 `Vector` 临时变量而不是 `List`。

2.2 减点算法

Algorithm 1 removeMarkers

Input: `Vector<Point> &pts; Vector<bool> tag; tag` 的第一个和最后一个元素一定为 `false;`

Side effect: `pts` 中 `tag[i]` 为 `true` 的对应点 `pts[i]` 将被删除;

```
1: 第一个点一定不会被删，从第二个点开始;
2: int count = 1;
3: for (int i = 1; i<pts.size(); i++) do
4:     if tag[i]==false then
5:         if count!=i then
6:             pts[count] = pts[i];
7:         end if
8:         count++;
9:     end if
10: end for
11: pts.resize(count);
```

如算法1所示。在这里我们在 `pts` 原址进行修改，因为减点后的示踪点列不会比原点列更长。我们用后面要保留的点覆盖掉前面要被删除的点的内存，最后 `resize` 一下就可以了。对

于每个留下来的示踪点至多调用了一次拷贝构造函数，而对于删除的节点则什么都不需要做。

2.3 加点算法

Algorithm 2 addMarkers

Input: `Vector<Point> &pts; Real lowBound;`

Side effect: 在 `pts` 中距离过长的相邻点间加点；

```
1:  Vector<Point> res(2 * pts.size());
2:  res[0] = pts[0];
3:  int count = 1;
4:  for (int i = 0; i<pts.size()-1; i++) do
5:      if norm(pts[i+1]-pts[i])>lowBound then
6:          localpts = 需要添加的点列;
7:          for auto &lpt : localpts do
8:              res[count] = lpt;
9:              检查 res 容量是否已满，满则扩容;
10:             count++;
11:         end for
12:     end if
13:     res[count] = pts[i+1];
14:     检查 res 容量是否已满，满则扩容;
15:     count++;
16: end for
17: res.resize(count);
18: pts = res;
```

如算法2所示。在这里我们首先建立了一个 `Vector` 临时变量，并且分配了比较多的内存，用来存储加点后的示踪点列。对于每个示踪点我们调用了一次拷贝构造函数，并做了一次是否即将越界的判断。而对于每个新加入的点，插入的时间复杂度是 O(1) 的。

这样我们就实现了既不建立 `List` 临时变量，对每个点的插入和删除的时间复杂度又控制在了 O(1) 内。代价是不论是否真的需要加点，在 `split` 函数中都要创建临时的 `Vector` 变量并调用拷贝构造。

3 测试结果

以下两个测试都是针对 Cubic MARS 方法的即殷集边界是用三次样条曲线表示的。测试中首先在某一个圆上均匀取点，之后由这些点生成的三次样条曲线作为初值。h_L 取值为两个初值点间弧长的 2 倍。在 CPU 时间这部分，我们对三种方法进行了测试，第一种是用 `Vector` 容器的，第二种是用 `List` 容器的，第三种使用上一节中的 IMV 的，结果取 10 次测试的平均值。

3.1 Vortex shear of a circular disk

速度场如下：

$$\begin{cases} u_x = \cos\left(\pi\dfrac{t}{T}\right)\sin^2(\pi x)\sin(2\pi y); \\ u_y = -\cos\left(\pi\dfrac{t}{T}\right)\sin(2\pi x)\sin^2(\pi y). \end{cases} \tag{6}$$

测试所用参数如表1所示。

参数	值
周期	$T = 8$
中心点	$C = (0.5, 0.75)$
半径	$R = 0.15$
r_{tiny}	$r_{\text{tiny}} = 0.01$
初值点数	$n = 64, 128, 256, 512, 1024$
时间步长	$k = 4\text{e}-2, 2\text{e}-2, 1\text{e}-2, 5\text{e}-3, 2.5\text{e}-3$

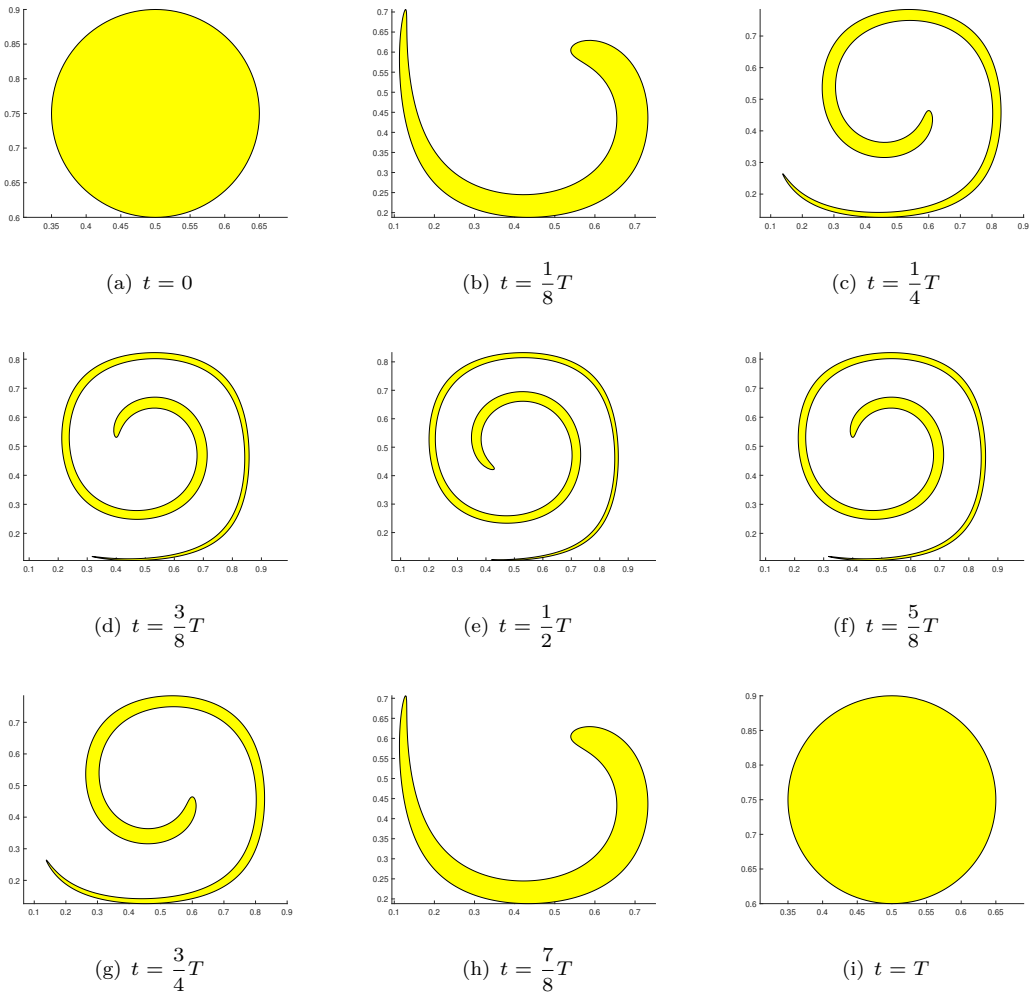
表 1: Vortex shear: 参数表

测试结果如表2所示，所用误差范数 $\|\text{E}\|_1$ 是用计算出的三次样条曲线和用准确解正圆上的点生成的三次样条曲线求内部区域间近似异或面积得出的。发现可以测得四阶以上的收敛阶。

中间步的计算结果图如图1所示。

$h_L = 4\pi/n$	$n = 64$	ratio	128	ratio	256	ratio	512	ratio	1024
$\|\mathrm{E}\|_1$	4.80e-5	4.25	2.52e-6	4.96	8.08e-8	7.41	4.77e-10	4.09	2.80e-11
Vector: CPU time(s)	2.91e-1	1.87	1.06	1.95	4.11	1.99	1.64e+1	2.02	6.63e+1
List: CPU time(s)	3.13e-1	1.87	1.15	1.96	4.45	1.98	1.76e+1	2.01	7.11e+1
IMV: CPU time(s)	2.83e-1	1.92	1.07	1.95	4.14	1.99	1.64e+1	2.02	6.65e+1

表 2: Vortex shear: 误差、收敛阶及运行时间对比



(a) $t = 0$ (b) $t = \frac{1}{8}T$ (c) $t = \frac{1}{4}T$

(d) $t = \frac{3}{8}T$ (e) $t = \frac{1}{2}T$ (f) $t = \frac{5}{8}T$

(g) $t = \frac{3}{4}T$ (h) $t = \frac{7}{8}T$ (i) $t = T$

图 1: Vortex shear: 中间步计算结果图，所用参数为 $n = 128$，$k = 2\mathrm{e} - 2$，$r_{\mathrm{tiny}} = 0.01$。

3.2 Deformation of a circular disk

速度场如下：

$$\begin{cases} u_x = \cos\left(\pi\dfrac{t}{T}\right)\sin(n\pi(x+0.5))\sin(n\pi(y+0.5)); \\ u_y = \cos\left(\pi\dfrac{t}{T}\right)\cos(n\pi(x+0.5))\cos(n\pi(y+0.5)); \\ n = 4. \end{cases} \tag{7}$$

测试所用参数如表3所示。这里我调整了一下测例，因为在这一次的测试中我发现 $n = 128$，$k = 4\mathrm{e}-2$ 时计算出来的结果无法正确的逼近准确解正圆，所以将时间步长缩短了一半，发现所有计算解都可以很好的逼近准确解正圆。

参数	值
周期	$T = 2$
中心点	$C = (0.5, 0.5)$
半径	$R = 0.15$
r_{tiny}	$r_{\text{tiny}} = 0.01$
初值点数	$n = 128, 256, 512, 1024, 2048$
时间步长	$k = 2\mathrm{e}-2, 1\mathrm{e}-2, 5\mathrm{e}-3, 2.5\mathrm{e}-3, 1.25\mathrm{e}-3$

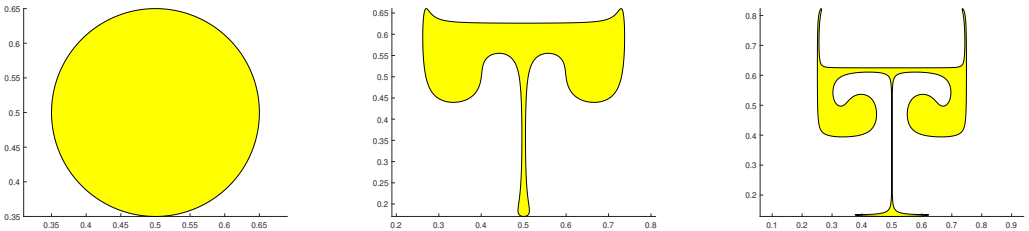
表 3: Deformation: 参数表

测试结果如表4所示，所用误差范数 $\|\mathrm{E}\|_1$ 是用计算出的三次样条曲线和用准确解正圆上的点生成的三次样条曲线求内部区域间近似异或面积得出的。发现可以测得四阶以上的收敛阶。

$h_L = 4\pi/n$	$n = 128$	ratio	256	ratio	512	ratio	1024	ratio	2048
$\|\mathrm{E}\|_1$	2.43e-5	3.67	1.91e-6	5.20	5.19e-8	6.56	5.50e-10	4.12	3.07e-11
Vector: CPU time(s)	3.01e-1	1.99	1.19	2.01	4.81	2.00	1.93e+1	1.84	6.92e+1
List: CPU time(s)	3.26e-1	1.97	1.28	2.01	5.13	1.99	2.04e+1	1.84	7.29e+1
IMV: CPU time(s)	3.04e-1	1.97	1.20	2.01	4.81	2.00	1.92e+1	1.84	6.90e+1

表 4: Deformation: 误差、收敛阶及运行时间对比

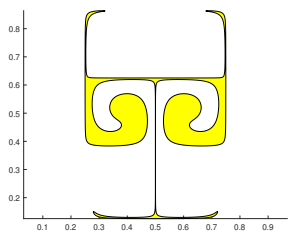
中间步的计算结果图如图2所示。



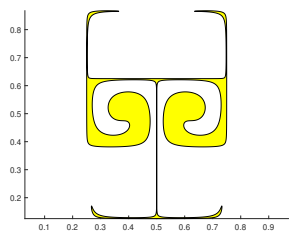
(a) $t = 0$

(b) $t = \frac{1}{8}T$

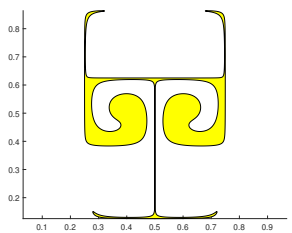
(c) $t = \frac{1}{4}T$



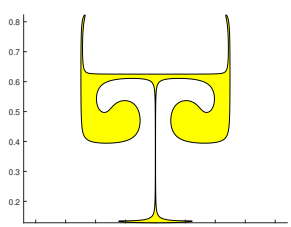
(d) $t = \frac{3}{8}T$



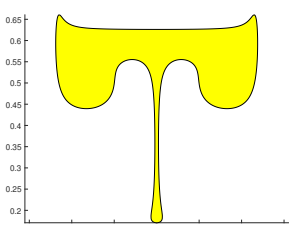
(e) $t = \frac{1}{2}T$



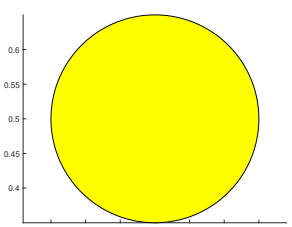
(f) $t = \frac{5}{8}T$



(g) $t = \frac{3}{4}T$



(h) $t = \frac{7}{8}T$



(i) $t = T$

图 2: Deformation: 中间步计算结果图，所用参数为 $n = 256$，$k = 2\mathrm{e} - 2$，$r_{\mathrm{tiny}} = 0.01$。

4 测试结果分析

从测试结果中可以看到，CPU 时间的变化阶数在 2 左右，除去时间步长每层缩短一半带来的影响，每一步的时间复杂度大约在 $\mathrm{O}(n)$，和我们的预期相同。

在 CPU 时间对比方面，发现 `List` 在所有情况下都要比 `Vector` 和 IMV 要慢，因为每一步都创建 `List` 临时变量所造成的计算时间增长掩盖了 `List` 加减点快带来的计算时间减小。而 `Vector` 和 IMV 相比 CPU 时间非常接近，IMV 相比于 `Vector` 优点在于加减点计算

量小，缺点在于即使没有进行加减点也要建立临时 `Vector` 变量和调用拷贝构造。从测试中我们看到随着示踪点数的增加，IMV 的 CPU 时间有时会低于 `Vector`，所以推测随着点数的增加和加减点的更加频繁，IMV 相比于 `Vector` 应该会有一定优势。