Генератор двумерных сеток с использованием алгоритма продвинутого фронта

Данилов Александр Анатольевич

13 апреля 2005 г.

1 Цель программы и ее основные характеристики

Основное назначение этой программы — генерация двумерных сеток областей, заданных своими границами. Программа состоит из основной библиотеки и нескольких примеров ее использования. Библиотека может использоваться в любых приложениях. Области могут быть как односвязными, так и многосвязными, однокомпонентными и многокомпонентными. Область должна быть ограниченной, и иметь кусочно-гладкую границу. Есть два основных способа задания области — аналитический и сегментный. Оба способа описывают только границу области.

Генератор создает конформную сетку. Конформная сетка это набор треугольников. Причем никакие два треугольника не пересекаются своими внутренними частями. Их объединение есть вся область. И для любых двух треугольников их пересечение — либо пусто, либо вершина, либо целое ребро.

Максимальное количество вершин, ребер и треугольников в сетке ограничено только объемом доступной оперативной памяти. Единственное необходимое для успешной генерации сетки условие на границу области — отсутствие самопересечений.

Для ускорения работы программы используются эффективные методы сортировки и структуры хранения данных. Библиотека написана на языке C, примеры работы с ней — на языке Fortran. Используемые типы данных: для целых чисел — 4-байтный integer, для вещественных чисел — 8-байтный double precision (в терминах Fortran'a).

2 Описание формата входных/выходных данных

2.1 Аналитический формат данных

Область считаем объединением непересекающихся подобластей. Граница подобласти — кусочно-гладкая, мы разбиваем ее на части, так чтобы каждая часть границы могла быть параметризована одной функцией. Такие части мы будем называть линиями. Конечные вершины линий мы назовем V-точками. Таким образом каждую линию можно описать следующим способом: указываются V-точки, где начинается и заканчивается данная линия, указывается параметризация линии, и в случае нетривиальной параметризации указывается интервал изменения параметра, на самом деле в последнем случае координаты V-точек можно вычислить явно, подставив в параметризацию границы интервала. Также необходимо еще указать, где именно находится область: справа, слева или по обе стороны от границы, в последнем случае границу будем назвать разрезом.

Перейдем теперь к формальному описанию данных. Функция для аналитического случая — `aft2dboundary`. Рассмотрим пример ее вызова:

```
      call aft2dboundary( Nbv, bv, Nbl, bl, bltail,  h,
     &           nv, vrt,
     &           nt, tri, material,
     &           nb, bnd,
     &           nc, crv, iFNC )
```

Исходные данные

Nbv и bv задают V-точки, Nbv — их количество, а в массиве $bv(2, Nbv)$ - координаты V-точек, которые нумеруются с 1 до Nbv. Nbl, bl и $bltail$ задают линии, Nbl — их количество, массив $bl(7, Nbl)$ содержит информацию о линиях, а массив $bltail(2, Nbl)$ задает интервалы изменения параметра для параметризованных линий. Таким образом на каждую линию приходится 7 целых чисел и 2 вещественных. Целые числа из bl означают следующее:

1. Номер V-точки, где начинается линия
2. Номер V-точки, где линия заканчивается
3. Число 0, если линия это просто отрезок, либо натуральное число, задающее параметризацию. Заметим, что пользователь сам должен написать функцию, отвечающую за параметризацию кривых. Это число передается функции пользователя, чтобы указать параметризующую функцию для данной линии.
4. Это целое число нигде не используется, и может принимать любое значение.
5. Цвет линии, этот параметр не влияет на процесс триангуляции. Все ребра этой линии в результирующей сетке унаследуют этот цвет.
6. Номер подобласти, граница которой содержит данную линию
7. Число 0, если линия является границей области, либо номер второй подобласти, если линия является разрезом. То есть два последних числа задают номера подобластей, которые находятся около этой части границы. Эти два числа могут совпадать, если разрез проходит внутри подобласти.

Пара вещественных чисел из $bltail$ имеет смысл для линий, у которых номер параметризации отличен от нуля. Эта пара задает интервал изменения параметра, причем первое число — для V-точки, где линия начинается, а второе — для той, где кончается. При этом в алгоритме координаты из массива bv не используются, а вычисляются с помощью параметризующей функции. То есть координаты таких V-точек могут принимать любые значения. Аналогично, для отрезков пара чисел из $bltail$ не имеет смысла, и может принимать любые значения.

Несколько правил для задания границы:

1. При движении вдоль линии от начальной точки до конечной подобласть должна находиться слева. Иначе говоря линии задают границу обходом против часовой стрелки.

2. Если линия является разрезом, то подобласть, номер которой записан в шестом числе, должна находиться слева.

3. Если разрез проходит внутри одной подобласти, то порядок следования конечных точек для линии не важен.

Вещественный параметр h задает желаемый размер треугольников. Генератор сделает квазиравномерную сетку с данным размером треугольников.

Параметризующая функция пользователя, должна быть задана в файле `crv_model.c`, и зарегистрирована с помощью `registeruserfn` до запуска генератора.

```
external userboundary
call registeruserfn( userboundary )
```

Результат работы

Результатом работы является сетка, которая задается набором вершин, треугольников и граничных ребер. nv и vrt задают вершины триангуляции, nv — количество вершин, а в массиве $vrt(2, nv)$ — их координаты. Все вершины пронумерованы с 1 до nv. nt, tri и $material$ описывают треугольники, nt — их количество, в массиве $tri(3, nt)$ хранятся номера вершин, образующих данный треугольник, в массиве $material(nt)$ хранится идентификатор подобласти, которой принадлежит треугольник. nb и bnd описывают граничные ребра, nb — их количество. Массив $bnd(4, nb)$ для каждого ребра содержит 4 целых числа. Первое и второе задают номера вершин данного ребра, третье число — 0, если ребро это отрезок, либо натуральное число, задающее номер этого ребра в списке криволинейных ребер, четвертое число - цвет ребра, унаследованный от границы. nc, crv и $iFNC$ описывают криволинейные ребра, nc — их количество, ребра в списке нумеруются с 1 до nc. В массиве $crv(2, nc)$ заданы значения параметра в начале и в конце ребра, а в массиве $iFNC(nc)$ — номера параметризующих функций.

2.2 Сегментный формат данных

Предполагается, что граница области уже разбита на отрезки, и требуется разбить саму область на треугольники так, чтобы след триангуляции на границе области в точности совпал с исходным разбиением. Таким образом на вход алгоритма подается набор отрезков, составляющих границу области.

Перейдем к формальному описанию данных. Функция для сегментного случая — `aft2dfront`. Рассмотрим пример ее вызова:

```
 call aft2dfront(
&           Nbr, brd, Nvr, vbr,
&           nv, vrt, cf,
&           nt, tri, material,
&           nb, bnd)
```

Исходные данные

Предлагается два способа описания граничных отрезков: явный (заданы вершины и заданы сами отрезки) и неявный (заданы только вершины обхода границы области). Внутри

генератора неявно заданная граница преобразуется в явно заданную, поэтому способ задания области не влияет на результирующую триангуляцию. Пользователь выбирает тот способ, который для него более удобен. В случае явного задания отрезков параметр $Nbr > 0$, а в случае неявного — $Nbr = 0$.

Nvr и vbr задают вершины, Nvr — их количество, в массиве $vbr(2, Nvr)$ их координаты. Вершины нумеруются с 1 до Nvr.

Если $Nbr > 0$, тогда Nbr — количество отрезков, а в массиве $brd(2, Nbr)$ номера начальных и конечных вершин для каждого отрезка. При этом порядок перечисления элементов как в списке вершин, так и в списке отрезков не важен. Единственное условие — при движении вдоль отрезка от начальной точки до конечной слева от отрезка должна находиться исходная область. То есть, если отрезок является разрезом, то он должен быть указан в списке дважды, сначала в одном направлении, а потом в другом.

Если $Nbr = 0$, то brd не имеет смысла и может принимать любое значение. В этом случае порядок вершин в исходных данных важен, и именно он заменяет недостающую информацию. Правила для задания области в этом случае:

1. Граница области разбивается на петли — замкнутые ломанные без самопересечений. Информация о каждой петле записывается в список вершин непрерывным блоком. Таким образом список вершин можно мыслить как несколько склеенных подряд списков, каждый из которых описывает одну петлю.

2. В списке, задающем одну петлю, первая и последняя вершины должны совпадать, то есть должны совпадать пары координат. Это свойство используется в генераторе для разбиения всего списка вершин, на списки для каждой петли.

3. Вершины петли заданы в списке в последовательном порядке, причем при обходе вдоль отрезков соединяющих соседние вершины слева должна лежать исходная область.

4. Петли могут касаться друг-друга произвольным образом, при этом общие вершины должны быть указаны в каждой из таких петель.

Вещественный параметр cf задает желаемый параметр разгрубления. Обычно этот параметр больше 1. Генератор сделает сетку, имеющую след на границе области в точности совпадающим с заданным разбиением границы области. При этом размер треугольников в сетке будет увеличиваться по мере удаления от границы области. Бóльшие значения cf будут давать более грубые триангуляции.

Так как граница области задается набором отрезков, то параметризация кривых в данном случае не нужна, не требуется ни написание параметризующей функции, ни ее регистрация.

Результат работы

Результат работы имеет тот же формат, что и в случае аналитического задания области. Так как в этом случае криволинейных граничных ребер не возникает, то nc, crv и $iFNC$ не используются, а в массиве $bnd(4, nb)$ третий столбец чисел не имеет смысла, и может принимать любые значения.

3 Основные идеи и используемые алгоритмы

Для успешной работы некоторых используемых алгоритмов необходимо, чтобы все вершины триангуляции лежали в квадрате $[0,1] \times [0,1]$. Для этого применяется сдвиговомасштабное преобразование координат, переводящее все вершины из входных данных внутрь единичного квадрата. Во время работы программы ни одна точка не должна покидать границы этого квадрата. Поэтому необходимо удостовериться, что не существует такого квадрата, который накрывает все точки, координаты которых указаны во входных данных, но при этом не накрывает всю область. Очевидно подобные проблемы могут возникнуть только в случае аналитически заданных областей, в этом случае можно явно указать прямоугольник, накрывающий всю область, и перечислить координаты его вершин в списке V-точек (на самом деле достаточно указать хотя бы две противоположные вершины)

Так как для работы алгоритма продвинутого фронта необходимо иметь набор отрезков, задающих границу области, то в случае аналитически заданной области, сначала происходит разбиение границы на отрезки с учетом желаемого размера треугольников. Если же область задана в сегментном формате, то никаких преобразований не требуется.

Для построения триангуляции используется алгоритм продвинутого фронта. Это итерационный алгоритм, строящий треугольники один за другим. Вначале рассматривается граница области и называется фронтом. Фронт это набор отрезков, образующий одну или несколько замкнутых ломанных, при этом для каждого отрезка определено направление, для того чтобы отличать в какой части плоскости относительно этого отрезка находится текущая не триангулированная область. На каждом шаге алгоритма выбирается один отрезок, и на нем, как на основании строится треугольник. Этот треугольник вырезается из области, и тем самым меняется фронт — из текущего фронта удаляются ребра, принадлежащие только что построенному треугольнику, и добавляются оставшиеся от треугольника ребра. Этот процесс продолжается до тех пор, пока в фронте есть хотя бы одно ребро. Получается, что на каждом шаге область Ω разбита на две части — Ω_Δ и Ω_0, где Ω_Δ это объединение всех построенных треугольников, то есть уже триангулированная часть области. Ω_0 — то, что осталось, еще не разбитая область. Фронт Γ в таком случае просто является границей области Ω_0. В самом начале работы алгоритма $\Gamma = \partial\Omega$, $\Omega_0 = \Omega$, $\Omega_\Delta = \emptyset$, а в конце работы алгоритма $\Gamma = \emptyset$, $\Omega_0 = \emptyset$, $\Omega_\Delta = \Omega$. Перейдем к детальному рассмотрению одного шага работы алгоритма.

На каждом шаге необходимо выбрать ребро из фронта и построить на нем, как на основании, треугольник. Треугольник должен удовлетворять требованиям пользователя, но при этом он должен целиком лежать внутри области, и не пересекаться с другими уже построенными треугольниками. То есть треугольник не должен пересекаться с текущим фронтом. Более строго условие для конформности сетки формулируется так: для двух произвольно выбранных отрезков, где один выбран из ребер треугольника, а другой из текущего фронта верно ровно одно из трех условий: либо они совпадают, либо не имеют общих точек, либо имеют ровно одну точку пересечения, причем для каждого отрезка эта точка — конечная вершина. Как только такой треугольник построен, происходит обновление фронта, и алгоритм переходит к следующему шагу. В предлагаемом алгоритме на каждом шаге из фронта выбирается ребро с наименьшей длинной, и на нем, как на основании, строится равнобедренный треугольник, высота которого определяется из желаемого размера. Заметим теперь, что у нового треугольника две вершины зафиксированы, а третью можно двигать, следя за тем, чтобы треугольник не пересекал фронт.

Текущего кандидата в качестве третьей вершины треугольника мы будем называть выделенной вершиной. Далее рассматривается окрестность выделенной вершины и если в нее попадают вершины из фронта, то среди них выбирается ближайшая и объявляется выделенной. Далее проверяется, не пересекает ли треугольник фронт, и если пересекает, то выбирается новая выделенная вершина — одна из вершин ребра, которое пересек треугольник. Этот процесс продолжается до тех пор, пока не найдется подходящий треугольник. При этом если было рассмотрено более 100 выделенных вершин, то процесс останавливается, и вызывается обработчик ошибок. Когда треугольник найден, мы добавляем выделенную вершину в список вершин триангуляции, если ее там до этого не было, добавляем треугольник в список треугольников, и обновляем фронт.

Когда триангуляция будет полностью построена, проводится несколько этапов улучшения параметров сетки. Каждая вершина немного сдвигается в определенном направлении так, чтобы углы около нее стали более одинаковыми. Вершины триангуляции, у которых много соседей разбиваются на пары вершин, а пары соседних вершин, у которых соседей слишком мало наоборот склеиваются. После чего сетка преобразуется обратным сдвигово-масштабным преобразованием в исходные координаты и возвращается пользователю.

Основные идеи ускорения работы программы

Рассмотрим наиболее трудоемкую часть работы — проверку на пересечение треугольником текущего фронта. Вспомним, что основание треугольника это ребро, уже лежащее в фронте, поэтому, надо проверить только две оставшиеся стороны. Для этого конечно можно было бы просто перебрать все ребра фронта, и для каждого проверить есть ли пересечение. Предлагается для хранения ребер фронта использовать структуру четверо-дерева. Четверо-дерево устроено как обычное бинарное дерево, только у каждой вершины не пара потомков, а четыре. Каждая вершина такого дерева будет отвечать за некоторую часть квадрата $[0,1] \times [0,1]$. При этом корень дерева отвечает за весь квадрат, а его потомки отвечает каждый за свою четверть исходного квадрата. И вообще при переходе от родителя к потомкам, квадрат родителя делится на 4 равные части, и распределяется между потомками. Уровнем вершины назовем количество ребер в пути от этой вершины до корня дерева, так у самого корня уровень будет 0, а у его потомков — 1. Теперь в каждой вершине будем хранить список приписанных к ней ребер из фронта. Чтобы выбрать какой вершине приписать ребро будем руководствоваться следующими правилами:

1. Вычисляем длину ребра l. Далее находим такое наименьшее число n, что $l \geqslant 2^{-n}$. Это число n будет означать уровень искомой вершины в четверо-дереве.

2. Находим середину ребра, и такую вершину в четверо-дереве на уровне n, которая отвечает за квадрат, содержащий найденную середину ребра.

Рассмотрим преимущества подобной структуры перед обычным списком. Пусть мы хотим проверить, пересекает ли данный треугольник Δ текущий фронт Γ. Для этого рассмотрим круг B, полностью покрывающий треугольник Δ, очевидно если ребро K_i из фронта Γ пересекает Δ, то оно пересекает и B. В фронте для каждого ребра K_i рассмотрим круг B_i, построенный на K_i, как на диаметре. Тогда если K_i пересекает B, то и B_i пересекает B. Очевидно, что B и B_i пересекаются тогда и только тогда, когда расстояние между их центрами меньше суммы радиусов, обозначим через ρ_i расстояние

от центра B до центра B_i (или что то же самое до середины K_i), пусть R — радиус B, R_i — радиус B_i, или половина длины ребра K_i. Тогда если ребро K_i пересекает Δ, то заведомо верно, что $\rho_i \leqslant R + R_i$. Заметим, что $R_i \leqslant 2^{-lev_i}$, где lev_i — уровень вершины, которой приписано ребро в четверо-дереве. Структура четверо-дерева позволяет очень быстро найти все такие ребра, для которых $\rho_i \leqslant R + 2^{-lev_i}$. Только для таких ребер есть смысл проверять пересечение с треугольником. Поэтому остальные ребра не проверяются.

4 Устойчивость алгоритма

Рассмотрим основные проблемы, которые могут возникнуть при работе данного алгоритма. Очевидно, проблемой для любого алгоритма может стать нехватка памяти, необходимой для работы, а также наличие машинной погрешности при выполнении операций с вещественными числами. Отметим, что предлагаемая реализация бережно относится ко всей выделяемой памяти, и освобождает ее после работы. Проблемы с машинной точностью почти полностью решены за счет использования более точных, хотя и трудоемких, алгоритмов.

Более подробно остановимся на ошибках, с которыми может столкнуться алгоритм при работе. Все возможные проблемы можно разделить на 4 группы:

1. Ошибки, возникающие при нехватке памяти.

2. Ошибки в самом алгоритме — например, нарушение структуры четверо-дерева. Если алгоритм написан правильно, и никаких проблем с памятью на компьютере нет, то эти ошибки никогда не должны появляться. Так за время тестирования ни одна подобная ошибка не проявилась.

3. Ошибки в исходных данных. Они возникают во время обработки исходных данных, еще до начала триангуляции. В этом случае надо убедиться, что входные данные корректны.

4. Некритические ошибки. Это подавляющее большинство остальных ошибок, например, на очередном шаге не удалось построить треугольник или обновить фронт. Такие ошибки появляются из-за геометрических особенностей областей, неудачно выбранных параметров разбиения, также они могут возникать из-за неправильной логики алгоритма, или его неправильной реализации.

Первые три группы ошибок для работы алгоритма заканчиваются полной остановкой. Четвертая группа ошибок передает управление обработчику ошибок, который пытается исправить положение. Как правило ошибки четвертой группы локальны, то есть они чувствительны к триангуляции в некоторой небольшой окрестности, а также зависят от параметра разгрубления. Поэтому можно попробовать изменить параметр разгрубления, и переразбить ту часть области еще раз. Для этого обработчик находит ребро, на котором алгоритм не может продолжить работу. Находит в уже построенной триангуляции все соседние с этим ребром треугольники, удаляет их, обновляет фронт, меняет параметр разгрубления и возвращает управление алгоритму триангулирования. Если же после многих неудачных попыток сетка так и не будет получена, то алгоритм выведет соответствующее сообщение и закончит работу, вернув пустую сетку.

Можно доказать, что для любой области алгоритм продвинутого фронта работает за конечное время. На входе алгоритма - конечный набор вершин, конечный набор отрезков,

образующий границу области, пусть S - площадь области, это конечная величина, а ρ_{min} - минимальное расстояние между вершинами. Таким образом длина любого отрезка будет не меньше ρ_{min}. Если мы докажем, что при триангулировании ни одна сторона нового треугольника не будет меньше ρ_{min}, то тогда очевидно на каждом шаге из области будет вырезаться треугольник с площадью не меньше $\rho_{min}^2\sqrt{3}/2$. То есть потребуется не больше чем $\frac{S}{\frac{\sqrt{3}}{2}\rho_{min}^2}$ треугольников.

Введем следующие обозначения: ρ_{min}^k - минимальное расстояние между уже построенными вершинами после шага k, ρ_{front}^k - минимальная длина отрезка в фронте после шага k. Тогда $\rho_{min}^0 = \rho_{min}$, $\rho_{min}^k \leqslant \rho_{min}^{k-1}$, $\rho_{front}^k \geqslant \rho_{min}^k$. На k-ом шаге мы выбираем минимальное ребро из фронта, его длина — ρ_{front}^{k-1}, и строим на нем равнобедренный треугольник с боковой стороной, равной $c\rho_{front}^{k-1}$, где c - текущий коэффициент разгрубления ($c \geqslant 1$). Далее если в окрестности третьей вершины треугольника есть вершины из фронта, то рассматривается расстояние до ближайшей — ρ^k, и если $\rho^k < \rho_{min}^{k-1}$, то третья вершина заменяется на вершину из фронта. Далее идет проверка треугольника на пересечение с фронтом. После чего возможны два случая:

1. Третья вершина треугольника совпала с одной из вершин фронта, тогда очевидно новых вершин не добавилось и $\rho_{min}^k = \rho_{min}^{k-1}$, при этом стороны треугольника не меньше ρ_{min}^{k-1}.

2. Третья вершина осталась без изменений, тогда стороны треугольника не меньше ρ_{min}^{k-1}. Добавилась новая вершина, но при этом расстояние от нее до любой другой не меньше ρ^k, и при этом по построению $\rho^k \geqslant \rho_{min}^{k-1}$, опять же $\rho_{min}^k = \rho_{min}^{k-1}$.

Таким образом $\rho_{min}^k = \rho_{min}$ и все стороны треугольников не меньше ρ_{min}, что и требовалось доказать.

Теперь рассмотрим более детально выбор не пересекающего фронт треугольника. Сначала строится равнобедренный треугольник, после чего его третью вершину мы начинаем менять до тех пор, пока треугольник не станет хорошим, то есть не пересекающимся с фронтом. Общая идея такова:

1. Если ребро выходит из одной из вершин треугольника, и при этом делит соответствующий угол, то выбираем другой конец этого ребра

2. Если треугольник пересекает ребро из фронта, причем в двух точках, то делаем третьей вершиной треугольника тот конец ребра, который лежит в той же полуплоскости относительно основания треугольника. Если оба конца лежат в нужной полуплоскости, то выбирается ближайшее к текущей вершине треугольника.

3. Если треугольник пересекает ребро из фронта, причем ровно в одной точке, то один его конец лежит внутри треугольника, а второй — снаружи. Выбираем тот, что лежит внутри.

4. Если ребро лежит внутри треугольника, то выбирается ближайший конец этого ребра

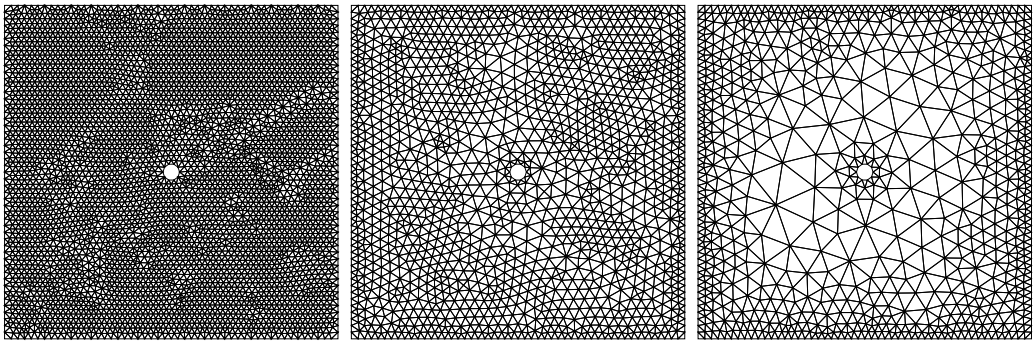
Очевидно, что при таком выборе третьей вершины треугольника, углы при основании будут не возрастать, более того, хотя бы один из них станет строго меньше, и, следовательно, этот процесс будет конечным. Однако у такого способа есть небольшой дефект —

после вырезания из области построенного треугольника может образоваться острый угол. Чтобы избежать подобных ситуаций дополнительно рассматриваются ребра, выходящие из основания треугольника, и проходящие рядом со сторонами треугольника. Третья вершина в этом случае перемещается в другой конец такого ребра, при этом запоминается, что на этом шаге мы увеличили угол при основании. Если в дальнейшем мы снова окажемся в такой же ситуации, то передвигать третью вершину мы не будем, чтобы не зациклиться. Таким образом почти всегда углы при основании будут уменьшаться, а увеличиваться будут лишь несколько раз, причем каждое увеличение будет происходить только один раз. Количество итераций можно грубо оценить количеством вершин, попавших в самый первый равнобедренный треугольник.

Подведем итог. Граница области должна быть замкнутой, правильно ориентированной, она должна ограничивать конечную область, и не должна самопересекаться. Это необходимое и достаточное условие для успешной работы алгоритма. Относительно работы программы, то есть конкретной реализации этого алгоритма, можно сказать только то, что указанное условие является необходимым, и почти всегда достаточным.

5 Примеры

В качестве примера области рассмотрим квадрат с вырезанным в центре кружочком. Мы задаем след триангуляции на границе квадрата и на окружности: 200 точек на квадрате и 10 точек на окружности. Все точки равномерно распределены на границе. Далее мы запускаем программу с тремя разными значениями параметра разгрубления c.

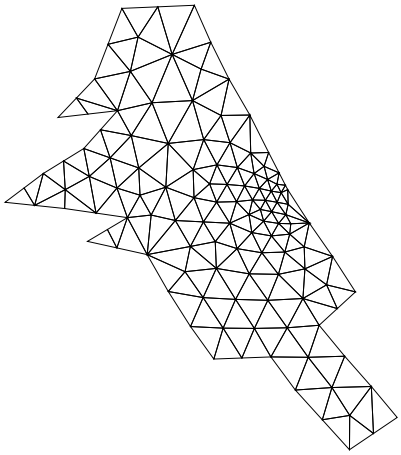


1. На первой картинке — результат при $c = 1.0$, количество треугольников $nt = 8770$, количество вершин $nv = 4490$. Все треугольники примерно одного размера. Использование коэффициента c, равного 1, целесообразно для получения квазиравномерной сетки, при условии, что след на границе также квазиравномерен.

2. Следующая картинка — $c = 1.2$. В полученной триангуляции треугольников и вершин гораздо меньше: $nt = 2828$, $nv = 1519$. Хорошо видно, что сами треугольники стали больше, хотя при этом стороны попавшие на границу области остались прежними.

3. Последняя картинка — $c = 1.5$. $nt = 1468$, $nv = 839$. Триангуляция возле самой границы области получилась такой, как в предыдущем случае, а внутри области — гораздо грубее. Большие значения коэффициента c полезны для получения грубой триангуляции.

Рассмотрим еще один пример. Область опять же задается следом на границе, при этом граница разбита на отрезки неравномерно — один из отрезков существенно меньше остальных. Параметр разгрубления $c = 1.3$, в результате — $nt = 195$, $nv = 122$.



Хорошо видно, что около маленького отрезка сетка более мелкая. При этом конечно в триангуляции возникают вытянутые треугольники, что неизбежно при заданном следе на границе.

Алгоритм очень удобен для триангулирования областей, полученных программным путем, так как достаточно только указать желаемый след на границе, и при этом не надо указывать функцию, отвечающую за желаемый размер треугольников в каждой точке области. Заметим также, что алгоритм справляется с неравномерно разбитой границей области, формат входных данных просто организован, и доступен один параметр, влияющий на процесс триангуляции. При этом остается возможность для пользователя достаточно просто задавать области с помощью аналитического формата данных.

6 Литература

1. George, P.L. (1996), Automatic Mesh Generation and Finite Element Computation, Handbook of Numerical Analysis, Vol. IV, 69-190.