Векторно-матричные вычисления в Arageli
НЕПОЛНЫЙ ЧЕРНОВОЙ ВАРИАНТ

С. С. Лялин

3 августа 2006 г.

1 Модель и представление

В библиотеке есть средства для представления векторов и матриц с любым типом элементов. Это шаблонные классы *vector* и *matrix*, которые объявлены в <arageli/vector.hpp> и <arageli/matrix.hpp> соответственно.

Класс *vector*, объявленный как

template <**typename** T, **bool** $REFCNT$ = **true**> **class** *vector*;

служит для представления объектов из множества T^n:

$$T^n = \{(a_0, a_1, \ldots, a_{n-1}) \mid a_i \in T,\ i = 0, \ldots, n-1\}, \qquad n \in \mathbf{Z}_+,$$

где T — это любой законченный тип; ограничения на него накладываются только операциями над вектором. Соответственно, класс *matrix*, который объявляется

template <**typename** T, **bool** $REFCNT$ = **true**> **class** *matrix*;

представляет объекты из $T^{m\times n}$:

$$T^{m\times n} = \{\, ((a_{0,0}, \ldots, a_{0,n-1}), \ldots, (a_{m-1,0}, \ldots, a_{m-1,n-1})) \mid \\ a_{i,j} \in T,\ i = 0, \ldots, m-1,\ j = 0, \ldots, n-1 \,\}, \qquad m, n \in \mathbf{Z}_+,$$

где на T, так же как и для векторов, накладываются ограничения только выполняемыми операциями над матрицей.

Заметим, что в представленных определениях, измерения объектов могут быть нулевыми, например, вектор с нулевым размером или матрица с n или m, или обоими вместе равными нулю. К тому же реальные

ограничения на размеры сверху определяются конкретной средой исполнения.

Для матрицы число m называется числом строк, а n — числом столбцов. Вектор не имеет ориентации и может быть как столбцом, так и строкой в операции (такие операции, где возникает вопрос об ориентации вектора, возникают только при смешанных вычислениях с матрицами).

Второй параметр шаблона $REFCNT$ контролирует включение подсчёта ссылок. В основном, результат набора операций, который применим для векторов и матриц, не зависит от этого параметра. О тех операциях, для которых это не выполняется, будет сказано особо. По умолчанию *vector* и *matrix* используют механизм подсчёта ссылок ($REFCNT$ **= true**).

Для хранения элементов данные классы используют один непрерывный кусок памяти, в котором в каждый конкретный момент времени есть как занятое пространство под элементы, так и не занятое, резервное пространство, размер которого, впрочем, может быть нулевым. Все операции, требующие изменения размеров (т.е. параметров m и n), могут проходить с перевыделением нового куска памяти, или полностью осуществляться в старой области, в зависимости от того, помещается ли финальный полезный размер (n для векторов и mn для матриц) в используемый кусок памяти или не помещается.

Матрицы хранятся развёрнутыми по строкам. Для векторов гарантируется, что все элементы будут находиться в непрерывной области выделенного куска, т. е. элементы хранятся без “дыр”. В случае с матрицами это гарантируется только для элементов каждой строки в отдельности, т. е. между строками матрицы могут быть не занятые (а следовательно, не инициализированные) промежутки.

Далее, если не сказано особо, все размеры выражаются в размерах типа T, т. е. в **sizeof**(T). Размер полезной области, которая содержит элементы вектора/матрицы без дыр, будет называться просто размером вектора или матрицы, $size$, т. е. $size = n$ или $size = mn$ для векторов и матриц соответственно. Размер всего выделенного куска памяти в представлении будет называться полной резервной ёмкостью объекта, $capacity$. Очевидно

$$size \leq capacity.$$

Заметим, что для матриц может быть $size = 0$, но число сток или число столбцов быть не нулевым. Это вырожденный случай, но объекты с такими характеристиками могуть реально существовать.

2 Внутренние определения типов и констант

Внутри классов *matrix* и *vector* определено несколько типов, которые служат для представления различных характеристик, а так же делают удобным доступ к параметрам шаблона:

- *value_type*, *element_type* — Тип элементов; совпадает с типом T в определении шаблона. Имя *value_type* введено как синоним *element_type* для того, что бы вектор или матрицу можно было рассматривать как контейнер STL.

- *size_type* — Представляет размеры и индексы. Не нужно думать, что любое число, которое возможно задать этим типом, будет корректным индексом или значением размера. Разрешены только те значения, которые представимы *difference_type*.

- *difference_type* — Означает тип результата разности двух итераторов; знаковый тип. Таким образом мощность этого типа является ограничением на максимальное число элементов в векторе или матрице. Скорее всего, *difference_type* будет знаковым аналогом *size_type* (на подобие отношения пары **signed int** – **unsigned int**).

- *reference* — Тип ссылки по которой можно модифицировать объект; объекты этого типа возвращаются некоторыми модифицирующими операциями и итераторами, которые предоставляют прямой доступ к элементам. Имеет тот же интерфейс, что и *value_type*&.

- *const_reference* — Тип ссылки по которой нельзя модифицировать объект; возвращается при прямом доступе к элементам без права изменения некоторыми константными методами и итераторами. Имеет тот же интерфейс, что и **const** *value_type*&.

- *pointer* — Тип указателя на элемент, используется итераторами там же, где и *reference*. Полный доступ с возможностью модификации. Интерфейс *value_type**.

- *const_pointer* — Указатель на элемент, используется итераторами там же, где и *const_reference*. Доступ только по чтению. Интерфейс **const** *value_type**.

- *iterator* — Итератор для элементов; полный доступ.

- *const_iterator* — Итератор для элементов; только чтение.

Каждый из классов определяет эти имена типов в качестве членов способом наиболее подходящим для реализации. Все эти названия позаимствованы из STL, потому что и вектор, и матрица, если на них посмотреть как на контейнеры, ничем не отличаются от стандартных контейнеров STL.

Итераторы *iterator* и *const_iterator* предназначены для обхода элементов вектора и матрицы. Элементы вектора перебираются с начала к концу, элементы матрицы — построчно слева направо, сверху вниз. Более подробно об итераторах будет сказано в соответсвующей главе.[1]

В добавление к типам, определен один статический член:

static const bool *refcounting* = $REFCNT$;

который позволяет получить значение признака включён/выключен для счётчика ссылок, т. е. имеет значение второго параметра шаблона.

3 Инициализация

Для векторов и матриц реализован большой набор конструкторов, которые позволяют инициализировать объекты почти любым вообразимым способом. Функциональность каждого конструктора дублируется соответствующей функцией из семейства *assign*, которая делает то же, что и конструктор, но может быть вызвана в любой момент времени жизни объекта, т. е. позволяет осуществить переинициализацию сходным образом.

Из-за большого числа конструкторов неизбежно возникает вопрос автоматического выбора какого-то конкретного из них при компиляции. Как известно, вызов конкретной версии перегруженного конструктора в C++ определяется только типами его фактических параметров. К сожалению, конструкторов у классов *vector* и *matrix* так много, что некоторые вызовы приводят к неоднозначности. Этой же неоднозначности способствует и принцип смешанных вычислений, которым пользователи библиотеки не стесняются пользоваться.

Для того, что бы избежать проблем при вызове конструктора, введены дополнительные параметры специальных типов, которые служат только для того, что бы вызвать нужную версию перегруженного конструктора. К тому же задействуется механизм определения связи между

[1] В момент написания этого документа множество итераторов, которые доступны для векторов и матриц, находится в разработке. Перечисленные классы (*iterator* и *const_iterator*) предоставляют всего лишь один способ обхода из спектра возможных. В скором времени появятся и другие итераторы.

типами и категоризация типов на этапе компиляции. Более подробно о том, как реализованы смешанные вычисления в Arageli описывается в соответсвующем документе.

3.1 Схема сигнатур конструкторов

Существует достаточно большое число конструкторов. Каждый конструктор в отдельности интереса не представляет, так как, всё же, это детали реализации. К тому же реализация усложняется требованиями механизма смешанных вычислений.

Здесь мы будем рассматривать *методы инициализации* и разберём, по какому принципу строются конструкторы, что бы каждый из этих методов был доступен пользователю в максимально удобном и простом виде.

При инициалиации как вектора, так и матрицы мы явно или не явно указываем два свойства:

- Размеры создаваемого объекта: число элементов в векторе, число строк и столбцов в матрице. По умолчанию размеры нулевые, т. е. если размеры явно не указаны каким-либо из доступных способов, то создаётся объект без элементов с нулевыми размерами $size = n = m = 0$.

- Значения, которыми будут заполняться создаваемые объекты (инициализационные значения). Значение по умолчанию — это нуль, точнее: *factory*$<T>$ *::* *null*().

Значения для инициализации могут быть представлены в трёх формах:

- Один объект val типа конвертируемого в T через вызов конструктора. Т. е. это такой тип Val для которого выполнено условие *type_pair_traits*$<Val, T>$ *::* *is_initializable* $==$ **true**.

- Итератор на начало последовательности из элементов конвертируемых в T. В последовательности должно быть достаточно элементов: информация о конце последоваетльности в виде итератора не принимается.

- Наконец, вектор или матрицу можно инициализировать другим вектором или матрицей, может быть, даже не являющимся объектом класса *vector* и *matrix*.

Набор реализованных конструкторов был бы труден для запоминания, если бы сигнатуры конструкторов не подчинялось некоторому правилу. А такое правило существует. Сигнатура любого конструктора удовлетворяет следующей схеме:

$$class\ (sizes,\ values,\ helpers), \tag{1}$$

где $class$ — это либо *vector*, либо *matrix*, $sizes$ — от нуля до двух параметров типа *size_type* определяющие размер создаваемого объекта, $values$ — один параметр с инициализационными значениями, $helpers$ — дополнительные вспомогательные параметры, помогающие вызвать нужный пользователю конструктор. Каждая из групп параметров $sizes$, $values$ и $helpers$ может быть пустой.

Группа параметров $helpers$ введена потому, что порою по параметрам первых двух групп невозможно правильно определенить нужный метод инициализации. Параметры этой группы через перегрузку позволяют отличать один конструктор от другого, если этого нельзя сделать автоматически.

Для векторов $sizes$ всегда состоит только из одного параметра, для матриц же если он содержит только один параметр, то создаётся квадратная матрица, если два — то матрица произвольных размеров, причём первое число в $sizes$ — это число строк, а второе — число столбцов.

Типы параметры $helpers$ находятся среди множества вспомогательных типов, которым, как указывалось выше, соответствуют вспомогательные константы:

- *fromval* — интерпретирует параметр из $values$ как значение;
- *fromseq* — интерпретирует параметр из $values$ как итератор на начало последовательности значений;
- *nonsquarte* — указывает на то, что в группу $sizes$ входят два параметра, первый означает число строк, второй — число столбцов;
- *colwise* — предписывает заполнять матрицу по столбцам из последовательности;
- *diag* — диагональная матрица;
- *eye* — единичная матрица.

Причём, при указании в составе $helpers$ параметры *fromval* и *fromseq* идут последними.

3.2 Общие конструкторы

Они есть в любом другом классе Arageli: конструктор по умолчанию, конструктор, преобразующий из строки, и конструктор смешанного копирования. Далее приведены объявления только для *vector*; для матриц объявления абсолютно идентичны, за исключением того, что везде вместо *vector* нужно поставить *matrix*.

```
template <typename T, bool REFCNT = true> class vector {
    ...
    /* (2) */
    vector ();

    /* (3) */
    vector (const char* s);

    /* (4) */
    template <typename T2, bool REFCNT2>
    vector (const vector<T2, REFCNT2>& x);
    ...
};
```

Конструктор по умолчанию (2) создаёт объект с $size = n = m = 0$[2]. Конструктор (3) инициализирует объект из строки. Правила записи вектора или матрицы в строке соответствуют формату, который по умолчанию использует система ввода/вывода представленная операторами **operator>>** и **operator<<**. Элементы вектора, разделяемые запятыми, записываются в строку и вся запись обрамляется круглыми скобками. Каждая строка матрицы представляется так же как вектор, а вся матрица записывается как вектор строк. Например, следующие объекты

$$v = \begin{pmatrix} 1 \\ -1 \\ 2 \\ -2 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \end{pmatrix},$$

$$y = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & x-2 \\ x^3-17 & 2x^5+1 \end{pmatrix}$$

из которых v мы представим в виде *vector*, а все остальные как *matrix*, могут быть записаны так

[2]Везде далее, когда речь идёт сразу и о векторах, и о матрицах, m будет иметь смысл только для матриц.

```
vector<big_int> v = "(1, -1, 2, -2)";
matrix<rational<> > x = "((1), (1/2), (1/3), (1/4))";
matrix<rational<> > y = "((1, 2, 3, 4))";
matrix<sparse_polynom<int> >
    A = "((1, x-2), (x^3-17, 2*x^5+1))";
```

Конструктор (4) служит для преобразования вектора или матрицы с разными параметрами шаблона друг в друга. Преобразование происходит поэлементно с вызовом соответствующего преобразующего конструктора для каждого из элементов. Данный конструктор является частью механизма смешанных вычислений и позволяет осуществить любые разумные с точки зрения пользователя преобразования.

3.3 Нулевые объекты с заданными размерами

Для создания объекта с заданными произвольными размерами и нулевыми значениями всех элементов используются следующие конструкторы:

```
template <typename T, bool REFCNT = true> class vector {
    ...
    /* (5) */
    explicit vector (size_type n);
    ...
};

template <typename T, bool REFCNT = true> class matrix {
    ...
    /* (6) */
    explicit matrix (size_type m);

    /* (7.a) */
    matrix (size_type m, size_type n);

    /* (7.b) */
    matrix (size_type m, size_type n, const nonsquarte_t& select);
    ...
};
```

Каждый элемент структуры инициализируется значением *factory*<T>*::null*(). Размер создаваемого вектора задаётся n (конструктор (5)), а для размеров матрицы либо $m = n$ (квадратная матрица, конструктор (6)), либо величины m и n задаются отдельно (матрица с любыми размерами, конструкторы (7.a) и (7.b)). Тип *size_type* подобно STL определяется

внутри классов матрицы и вектора и служит для представления размеров.

На самом деле, конструкторы (7.a) и (7.b) выполняют одну и ту же работу. По сути, это один и тот же конструктор в двух формах. Форма (7.b) введена для того, что бы разрешить появляющиеся неоднозначности при вызове конструкторов с двумя параметрами для матриц.

Параметр *select* — это фиктивный параметр, тип которого определяет, какой из перегруженных конструкторов с тремя параметрами (они будут рассмотрены ниже) нужно вызвать. Тип *nonsquarte_t* позволяет выбрать именно конструктор (7.b) а ни какой-либо другой. При вызове нужно пользоваться константой *nonsquarte*, которая имеет тип *nonsquarte_t* и определена в <arageli/matrix.hpp>:

```
struct nonsquarte_t {};
const nonsquarte_t nonsquarte =
    nonsquarte_t();
```

Все остальные вспомогательные типы и соответствующие константы, о которых будет сказано далее имеют определения подобные данному.

Следующий код, использующий конструкторы (5)–(7):

```
vector<int> x(3);
matrix<int> a(2);
matrix<int> b(2, 3, nonsquarte);
```

задаёт такие объекты:

$$x = (0,0,0)\,, \quad a = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Заметим, что в данном примере матрицу b нельзя задать так:

```
matrix<int> b(2, 3);
```

потому что, это не будет вызов конструктора (7.a). Почему это так, будет ясно после того, как будут рассмотрены другие конструкторы. Вместо этого можно написать так:

```
matrix<int> b(2, size_t(3)); // вызов (7.a)
```

если есть гарантия, что *matrix*<**int**>*::size_type* и *size_t* — это один и тот же тип (что почти наверняка так). После рассмотрения остальных двухаргументных конструкторов для матриц будет видно, что (7.a) можно вызвать только в том случае, если тип фактического параметра совпадает с соответствующим типом в сигнатуре конструктора, т. е. с *matrix*<**int**>*::size_type*.

3.4 Инициализация заданными значениями

Данная группа конструкторов позволяет задать любые размеры для вектора или матрицы и присвоить их элементам произвольные значения. Значения можно задать двумя способами. Либо это один элемент определённого типа, значение которого получит каждый элемент структуры. Такой метод инициализации схож с конструкторами предыдущей группы в том, что задаёт одно и тоже значение для всех элементов (в рассмотрении выше это был нулевой элемент *factory*<T>*::null*(), значение которого явно не передавалось). Либо инициализационные значения подаются в виде произвольной последовательности, представленной итератором на начало. Это важный элемент интерфейсов вектора и матрицы, так как позволят с лёгкостью использовать информацию из других частей библиотеки или из кода, который не является частью Arageli.

Здесь как раз и возникает проблема: как отличить один способ задания от другого? Ведь и в первом и во втором случае мы имеем одним из параметров конструктора объект произвольного типа, но должны обходится с ним по-разному в зависимости от того, что он из себя представляет. В первом случае это элемент-значение, во втором — итератор на начало последовательности. Очевидно, решение данного вопроса видится в той же схеме, по которой построен конструктор (7.b), т. е. нужно предоставить несколько вспомогательных типов, которые помогут выбрать нужный конструктор. И такое решение реализовано.

Но, есть и другой способ, который работает в большинстве случаев встречающихся на практике и не требует задания дополнительных параметров-селекторов. Он связан с использованием дополнительной информации о типе, а точнее о паре типов: первый — тип элементов T, над которым построен вектор или матрица, второй — тот неизвестный тип X, который представляет либо инициализационное значение, либо итератор на последовательность таких значений. Решение о природе X принимается очень просто: если X можно конвертировать в T, то X — это значение, в противном случае X — это итератор. Вопрос о конвертируемости решается с помощью *type_pair_traits*<X, T>. Шаблонная структура *type_pair_traits* определена в <arageli/typetraits.hpp>, предоставляет подобную информацию о любой паре типов и является частью механизма смешанных вычислений.

Далее перечислены определения конструкторов, в которых используются описанные идеи.

```
template <typename T, bool REFCNT = true> class vector {
    ...
    /* (8) */
    template <typename Val>
        vector (size_type n, const Val& val, const fromval_t& select);

    /* (9) */
    template <typename In>
    vector (size_type n, In in, const fromseq_t& select);

    /* (8+9) */
    template <typename X>
    vector (size_type n, const X& x);
    ...
};

template <typename T, bool REFCNT = true> class matrix {
    ...
    /* (10) */
    template <typename Val>
        matrix (size_type m, const Val& val, const fromval_t& select);

    /* (11) */
    template <typename In>
        matrix (size_type m, In in, const fromseq_t& select);

    /* (10+11) */
    template <typename X>
        matrix (size_type m, const X& x);

    /* (12) */
    template <typename Val>
        matrix (size_type m, size_type n,
        const Val& val, const fromval_t& select);

    /* (13) */
    template <typename In>
        matrix (size_type m, size_type n,
        In in, const fromseq_t& select);

    /* (12+13) */
    template <typename X>
        matrix (size_type m, size_type n, const X& x);
    ...
};
```

В этих определениях In обозначает тип итератора, Val — тип значения, которым будет инициализироваться каждый элемент, X — тип, для которого реализация выяснит, является ли он значением или итератором. Конструкторы с параметром X имеют пометку в виде $(A + B)$, что означает, что они обладают функциональностью конструкторов (A) или (B), в зависимости от того, каким будет результат решения вопроса о природе типа X.

Вспомогательные типы *fromseq_t* ("from sequence" — "из последовательности") и *fromval_t* ("from value" — "из значения") делают различие между (8) и (9), (10) и (11), (12) и (13).

Конструкторы (10), (11) и (10+11) служат для создания квадратных матриц, тогда как (12), (13) и (12 + 13) позволяют задать произвольное число строк и столбцов. Например, после выполнения следующего кода:

```
int array[] = {1, 2, 4, 8, 16, 32};

vector<big_int> x(2, &array[0]);
    // то же: x(2, &array[0], fromseq)

vector<int> y(2, rational<>(7));
    // то же: y(2, rational<>(7), fromval)

matrix<int> a(2, &array[0]);
    // то же: a(2, &array[0], fromseq)

matrix<rational<int> > b(2, 9);
    // то же: b(2, 9, fromval)

matrix<rational<big_int> > c(3, 2, &array[0]);
    // то же: c(3, 2, &array[0], fromseq)

matrix<rational<int> > d(3, 2, rational<int>("1/7"));
    // то же: d(3, 2, rational<int>("1/7"), fromval)
```

мы получим следующие объекты:

$$x = (1, 2)\,, \quad y = (7, 7)\,, \quad a = \begin{pmatrix} 1 & 2 \\ 4 & 8 \end{pmatrix},$$

$$b = \begin{pmatrix} 9 & 9 \\ 9 & 9 \end{pmatrix}, \quad c = \begin{pmatrix} 1 & 2 \\ 4 & 8 \\ 16 & 32 \end{pmatrix}, \quad d = \begin{pmatrix} 1/7 & 1/7 \\ 1/7 & 1/7 \\ 1/7 & 1/7 \end{pmatrix}$$

Обратите внимание на то, что тип элементов у вектора и матрицы варьируется и не всегда совпадает с типом тех значений, которыми инициализируются объекты. Но, тем не менее, благодаря механизму смешанных вычислений преобразования проходят корректно, и так же корректно определяется нужный конструктор. На самом деле, потребность в дополнительных селективных параметрах *fromval* и *fromseq* возникает в *очень* редких ситуациях. Но, всё-таки, если нельзя сделать никаких предположений насчёт типов, лучше их использовать.

В рассмотренных конструкторах, как это видно из примеров, при инициализации из последовательности элементы матрицы обходятся построчно, слева на право и сверху вниз. Причём, реализация гарантирует, что **operator*** для итератора *in* будет вызываться последовательно для элементов в последовательности и только по одному разу для каждого элемента. К тому же значение, полученное от итератора, будет сразу же использовано для инициализации и на него не будет сделано временной ссылки.

Последнее обстоятельство позволяет эффективно инициализировать матрицу не только хранящимися где-то элементами, но и генерируемыми на лету, например, как результат работы некоторой функции. Нужно только представить генератор значений в виде итератора. *[Здесь нужен пример!]*

Кстати, теперь понятно почему для вызова (7.*a*) требуется точное соответствие типа второго параметра. Если типы не будут совпадать, то это приведёт к вызову конструктора $(10 + 11)$ и вместо прямоугольной матрицы с нулевыми элементами мы получим квадратную матрицу со значениями равными второму (наверняка ненулевому) аргументу.

3.5 Дополнительные конструкторы для матриц

Реализовано несколько дополнительных конструкторов, которые позволяют создавать диагональные и, в частности, единичные матрицы. К тому же, наряду с инициализацией из последовательности по строкам (конструкторы (11) и (13)), есть инициализация по столбцам. Далее перечислены все дополнительные конструкторы:

```
template <typename T, bool REFCNT = true> class matrix {
    ...
    /* (14) */
    template <typename In>
        matrix (size_type m, In in, const colwise_t& select);
```

```
    /* (15) */
    template <typename In>
        matrix (size_type m, size_type n, In in,
        const colwise_t& select);

    /* (16) */
    template <typename Val>
        matrix (size_type m, const Val& val,
        const diag_t& select1, const fromval_t& select2);

    /* (17) */
    template <typename In>
        matrix (size_type m, In in,
        const diag_t& select1, const fromseq_t& select2);

    /* (16+17) */
    template <typename X>
        matrix (size_type m, const X& x, const diag_t& select1);

    /* (18) */
    template <typename Val>
        matrix (size_type m, size_type n, const Val& val,
        const diag_t& select1, const fromval_t& select2);

    /* (19) */
    template <typename In>
        matrix (size_type m, size_type n, In in,
        const diag_t& select1, const fromseq_t& select2);

    /* (18+19) */
    template <typename X>
        matrix (size_type m, size_type n, const X& x,
        const diag_t& select1);

    /* (20) */
    matrix (size_type m, const eye_t& select)

    /* (21) */
    matrix (size_type m, size_type n, const eye_t& select);
    ...
};
```

Для постолбцовой инициализации служат (14) и (15), первый для квадратных матриц, второй для матриц произвольных размеров. Столбцы заполняются сверху вниз и слева направо. Выполняются те же соглашения по использованию итератора in, что и при построчном заполнении.

Конструкторы (16)–(21) заполняют диагональ матрицы определённым значением; все остальные элементы приобретают нулевые значения *factory*<T>*::null*(). Для выбора некоторых конструкторов уже используется по два вспомогательных параметра ($select_1$ и $select_2$). При инициализации из последовательности диагональ проходится с левого верхнего элемента. (16), (17) и (16 + 17) создают квадратную диагональную матрицу, (18), (19) и (18 + 19) — матрицу произвольных размеров с заполненной диагональю, (20) и (21) — единичную матрицу: квадратную и общего вида.[3] При создании единичной матрицы диагональ заполняется значением *factory*<T>*::null*(). *[Здесь нужен пример!]*

4 Повторная инициализация и заполнение

В данную группу функций входят два семейства:

- Методы "повторной инициализации", которые позволяют задать значение объекту так же, как это можно сделать с помощью конструкторов, т. е. создать иллюзию повторного вызова конструктора (отсюда и название группы). Сюда входит операторы присваивания и семейство методов *assign*.

- Методы заполнения (семейство функций *fill*). Эти методы отличаются от предыдущей группы тем, что не изменяют размеров, а только присваивают элементам новые значения (так же, как это делают конструкторы).

Для каждого конструктора есть соответствующая функция *assign*. Более того, так как вспомогательные параметры для конструкторов введены только из-за того, что более нет способа выбрать перегруженный конструктор, то для семейства *assign* можно обойтись без них, введя несколько различных (не перегруженных) имён. Поэтому для некоторых конструкторов есть даже две функции *assign*.

Во-первых, для каждого конструктора есть функция с именем *assign* и с той же сигнатурой вызова, что и у конструктора (за исключением, конечно, типа возвращаемого значения: у *assign* — это **void**):

void *assign* ($sizes$, $values_{opt}$, $helpers_{opt}$),

Во-вторых, для каждой такой функции, у которой есть вспомогательные параметры $helpers$ существует дополнительная функция без них для

[3] Общепринято, что диагональные и, в частности, единичные матрицы являются квадратными. В Arageli можно создавать неквадратные аналоги этих видов матриц.

удобства написания. Имя этой функции начинается с "*assign* " и снабжено суффиксом, который содержит имена вспомогательных констант. Схематично это можно изобразить следующим образом. Пусть есть функция

void *assign* ($sizes$, $values_{opt}$, $helpers_{opt}$),

то для неё существует функция

void *assign_helpers* ($sizes$, $values_{opt}$),

которая выполняют ту же самую работу, что и первая. Можно вызвать любую из двух, — это зависит от предпочтений пользователя в конкретной ситуации. Результат будет одним и тем же. Например, для конструктора (17) определены следующие функции:

template <**typename** In>
void *assign* (*size_type* m, In in,
const *diag_t*& $select_1$, **const** *fromseq_t*& $select_2$);

template <**typename** In>
void *assign_diag_fromseq* (*size_type* m, In in);

а для (16+17) — такие:

template <**typename** X>
void *assign* (*size_type* m, **const** X& x, **const** *diag_t*& $select_1$);

template <**typename** X>
void *assign_diag* (*size_type* m, **const** X& x);

В отличие от *assign*, функции семейства *fill*определены не для каждого конструктора. Точнее говоря, одна функция зачастую соответствует нескольким конструкторам, т. к. нет необходимости делать разничие между квадратными матрицами и матрицами с произвольными размерами, ибо не нужно задавать размеры объекта: они остаются теми же.

Как и для *assign* есть две версии каждой *fill*-функции. Для конструктора в виде (1) это будет:

void *fill* ($values_{opt}$, $helpers_{opt}$),

и

void *fill_helpers* ($values_{opt}$).

Эти весьма удобные функции используются тогда, когда размеры вектора или матрицы не меняются, но нужно заполнить все элементы в стиле схожем с конструированием, что случается довольно часто.

Например, для рассмотренного выше конструктора (17) есть такие определения:

template <**typename** In>
void *fill* (In in, **const** *diag_t*& $select_1$, **const** *fromseq_t*& $select_2$);

template <**typename** In>
void *fill_diag_fromseq* (In in);

а для (16+17) — такие:

template <**typename** X>
void *fill* (**const** X& x, **const** *diag_t*& $select_1$);

template <**typename** X>
void *fill_diag* (**const** X& x);

[Здесь нужен пример: создали объект с помощью одного конструктора, вызвали assign — получили другое значение и т. п.]

5 Доступ к элементам

5.1 Обращение по индексу

5.2 Использование итераторов

6 Базовые алгебраические операции

7 Вставка и удаление

8 Манипулирование строками и столбцами

9 Размеры

10 Подвекторы и подматрицы