ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ ПРЯМЫХ МЕТОДОВ ЛИНЕЙНОЙ АЛГЕБРЫ

В.А. Семенов, О.А. Тарлапан

В данной статье представлены результаты применения объектно-ориентированной технологии к программированию задач и методов линейной алгебры. В предлагаемом подходе все вычислительно-конструктивные понятия линейной алгебры рассматриваются в качестве объектов и представляются единой классификационной иерархией. Наряду с широкими классами общих, специальных и элементарных матриц, иерархия включает классы самих задач и методов линейной алгебры. Известные объектные парадигмы применительно к разработанным матричным, проблемным и алгоритмическим классификациям обнаруживают важные инструментальные возможности для унифицированной программной реализации прямых методов линейной алгебры с использованием технологии элементарных матриц. Применение данного подхода представляется перспективным для эволюционной разработки математических библиотек и прикладных вычислительных систем, к которым предъявляются повышенные требования расширяемости, модифицируемости и адаптируемости.

1. Введение

В настоящее время объектно-ориентированный подход (ООП) с успехом применяется в самых разнообразных областях программирования, включая разработку математических библиотек и прикладных вычислительных систем [6]. Роль объектных технологий при реализации матричного программного обеспечения, предназначенного для решения задач линейной алгебры, также хорошо осознана и составляет предмет многочисленных публикаций. Проводимые исследования в основном концентрируются вокруг вопросов проектирования матричных классовых иерархий и выработки практических парадигм программирования методов линейной алгебры, прежде всего, как методов соответствующих матричных классов [6, 8].

Данный подход обеспечивает существенную программную общность, поскольку реализация методов решения новых классов линейных задач, отличающихся от имеющихся типами матричных объектов, сводится к созданию производных классов в рамках единой матричной иерархии. При этом основная часть

вычислительных методов непосредственно реализуется в общих матричных классах и автоматически наследуется всеми производными классами. В частных же классах переопределяется лишь часть методов (в основном базовые операции линейной алгебры типа BLAS 2, 3), реализация которых возможна или целесообразна с учетом специфических математических, вычислительных и программных особенностей вводимых матричных типов.

Вместе с тем, применяемый подход не может быть признан вполне удовлетворительным, особенно в тех случаях, когда требуется расширить или изменить алгоритмический репертуар матричной библиотеки. В таких ситуациях программирование новых методов линейной алгебры сводится к добавлению новых функций в интерфейс соответствующих матричных классов и к их реализации в традиционной процедурной манере.

Вполне естественным шагом при использовании ООП является рассмотрение самих ***методов*** и ***алгоритмов*** линейной алгебры как самостоятельных объектов в общей структуре математической объектно-ориентированной библиотеки [7]. Выделение алгоритмических объектов представляется достаточно перспективным для эволюционной разработки математических библиотек, алгоритмический набор которых может постепенно наращиваться, изменяться или адаптироваться к конкретным прикладным задачам. При этом существенно упрощается разработка самих библиотек, а их пользовательские характеристики улучшаются сразу в нескольких направлениях.

Во-первых, выделение классов линейных задач и методов их решения, наряду со вспомогательными векторными и матричными классами, позволяет более четко структуризовать программные средства, необходимые как для постановки, так и для решения задач. Подобное выделение помогает выразить традиционные математические понятия реальными программными объектами и в конечном итоге достичь желаемой наглядности и выразительности, позволяющей писать сложные прикладные программы в ясной и лаконичной нотации, близкой к математической.

Во-вторых, применение основных принципов ООП к разработанным проблемным, алгоритмическим и матричным классификациям обнаруживает целый ряд практических парадигм, которые могут рассматриваться в качестве мощной инструментальной основы для разработки как самой математической библиотеки, так и ее разнообразных приложений.

Предмет настоящей статьи составляет одна из парадигм, порождаемых данным подходом, а именно, парадигма

программирования прямых методов линейной алгебры с использованием библиотеки элементарных матричных классов. Как известно, прямые методы — будь то методы решения систем линейных алгебраических уравнений (СЛАУ), проблем собственных значений или задач о наименьших квадратах — теоретически базируются на тех или иных элементарных матричных преобразованиях, которые приводят задачу к эквивалентной, но более простой форме, допускающей ее непосредственное решение. Причем набор типов преобразований, необходимый для реализации большинства прямых методов, оказывается относительно небольшим. Данное обстоятельство послужило отправной точкой для исследования возможностей применения ООП к программированию прямых методов и разработки унифицированного подхода к их реализации с использованием технологии элементарных матриц.

В разделе 2 строится общая объектная классификация элементарных матриц, определяющих основные матричные преобразования. В разделе 3 прямые методы линейной алгебры рассматриваются как композиции матричных преобразований. При этом их программная реализация основывается на организации обобщенного суперкласса прямых методов, применимого ко всем конкретным типам задач линейной алгебры. Раздел 4 посвящен наиболее важному и широкому классу прямых методов — методов матричной факторизации.

2. Объектная классификация элементарных матриц

Прямые методы линейной алгебры, в конечном счете, редуцируются к сравнительно небольшому набору элементарных матричных преобразований [1–5]. Применение ООП позволяет представить данные преобразования как матричные операции с объектами особого рода — элементарными матрицами. Определение необходимого набора таких матриц и представление их классами единой матричной иерархии, наряду с основными функциональными матрицами, составляет важную методологическую часть проводимого подхода.

С этой целью в разработанной матричной классификации [8] выделяется особая группа элементарных матричных классов **ElementaryMatrix**, а для основных матричных объектов класса **GeneralMatrix** виртуально определяются необходимые арифметические бинарные операции с элементарными матрицами (операции матричного умножения определяются в двух вариантах, которые соответствуют левостороннему и правостороннему

умножению). В этом случае методы линейной алгебры, будучи редуцированными к элементарным преобразованиям, могут реализовываться уже на верхних уровнях матричной иерархии и автоматически наследоваться всеми частными классами. Тем самым достигается значительная общность в программной реализации методов решения задач, различающихся типами матричных объектов.

Создание конкретного матричного класса в рамках данного подхода сводится к реализации набора операций с элементарными матрицами, определяемого родительским классом **GeneralMatrix**. Заметим, что значительная часть подобных операций непосредственно реализуется в верхних матричных классах. Тем не менее, переопределение их в конкретных классах с учетом частных математических свойств, особенностей конкретных структур данных позволяет при необходимости добиться наибольшей эффективности кода. Таким образом, достигается важный компромисс между универсальностью разрабатываемого матричного обеспечения с одной стороны и его эффективностью с другой.

Ключевыми вопросами применения описываемого подхода являются определение набора элементарных матриц, необходимого для представления основных методов линейной алгебры, и организация соответствующей библиотеки классов. По-видимому, организация и состав библиотеки должны соответствовать набору базовых операций линейной алгебры, входящих в известный пакет BLAS 2, 3 и составляющих функциональную основу для процедурной реализации большинства программ линейной алгебры. При этом каждый создаваемый класс элементарной матрицы может быть выбран таким образом, чтобы быть функционально эквивалентным той или иной значимой группе базовых процедур BLAS 2, 3. Например, процедуры перестановки пар строк или столбцов матрицы представляются как операции умножения исходной матрицы на элементарную матрицу перестановок и т.п.

Рассмотрение теории матричного анализа и линейной алгебры [1–5] выявило следующий необходимый набор элементарных матриц, представленный единой иерархией матричных классов (рис. 1).

- **Matrix**
  - **GeneralMatrix**
  - **SpecialMatrix**
  - **ElementaryMatrix**
    - **IdentityMatrix**
    - **BackwardIdentityMatrix**
    - **DiagonalMatrix**
      - **ScalingMatrix**
    - **PermutationMatrix**
      - **TranspositionMatrix**
    - **MultiRankModificationMatrix**
      - **FrobeniusMatrix**
        - **FrobeniusColumnMatrix**
          - **FrobeniusColumnUpMatrix**
          - **FrobeniusColumnDownMatrix**
        - **FrobeniusRowMatrix**
          - **FrobeniusRowLeftMatrix**
          - **FrobeniusRowRightMatrix**
    - **OrthogonalElementaryMatrix**
      - **HouseholderMatrix**
      - **GivensRotationMatrix**
      - **JacobiRotationMatrix**
    - **BasicCirculant**
    - **ForwardToeplitz**
    - **BackwardToeplitz**

Рис. 1. Объектная классификация элементарных матриц

Вершиной матричной иерархии является абстрактный класс **Matrix**, обобщающий свойства всех матриц. Далее иерархия продолжается тремя основными классами **GeneralMatrix**, **SpecialMatrix** и **ElementaryMatrix**, объединяющими основные функциональные группы произвольных, специальных и элементарных матриц. Вопросы организации основных матричных классов рассмотрены в работе [8]. Остановимся более подробно на семействах элементарных матриц.

Класс **IdentityMatrix** представляет традиционный математический объект — единичную матрицу. Данный тип матрицы достаточно часто используется в вычислительной математике для представления более сложных матричных типов. Аналогично назначение класса **BackwardIdentityMatrix**, определяющего перъединичную матрицу.

Класс **DiagonalMatrix** определяет группу диагональных матриц, содержащих ненулевые элементы только на главной диагонали. Преобразования масштабирования, связанные с данной матрицей, часто применяются для предобуславливания матриц при решении линейных систем и балансировки матриц при решении задач на собственные значения. Иногда требуется промасштабировать только одну строку или один столбец. С этой целью в классификации определен класс матриц **ScalingMatrix**,

отличающихся от единичной каким-либо одним диагональным элементом.

Класс матриц перестановки **PermutationMatrix** определяет группу матриц, каждый столбец и строка которых содержат по одному единичному элементу. Данный матричный класс соответствует преобразованиям переупорядочения столбцов и строк в основной матрице, которые обычно применяются в методах выбора главного элемента, а также в методах структуризации и минимизации заполнения в разреженных матрицах. Элементарные матрицы перестановок, реализуемые классом **TranspositionMatrix**, представляют частный случай матриц перестановок, у которых только два ненулевых элемента расположены вне главной диагонали. Данный класс матриц определяет элементарную перестановку двух столбцов или двух строк.

Класс **FrobeniusMatrix** реализует семейство элементарных матриц Фробениуса — матриц единичного ранга, представимых формулой

$$\mathbf{F} = \mathbf{I} + \mathbf{x} \times \mathbf{y}^T,$$

где $\mathbf{I}$ — единичная матрица, $\mathbf{x}$ и $\mathbf{y}$ — два ненулевых вектора-столбца. Матрица данного типа является полной квадратной, однако, для ее эквивалентного представления достаточно хранить только элементы двух векторов. Данная матрица осуществляет одноранговую матричную модификацию и часто используется в прямых методах.

Вместе с ней употребимы также и ее частные варианты, порождаемые специфической разреженной организацией составляющих векторов. Данные варианты реализуются соответствующими матричными классами, которые наследуются от **FrobeniusMatrix**. В иерархии представлены классы **FrobeniusColumnMatrix**, **FrobeniusRowMatrix**, реализующие элементарные столбцовые и строчные матрицы Фробениуса, а также их варианты **FrobeniusColumnUpMatrix**, **FrobeniusColumnDownMatrix**, **FrobeniusRowLeftMatrix** и **FrobeniusRowRightMatrix**, соответствующие элементарным верхним и нижним столбцовым, а также левым и правым строчным матрицам. Перечисленные варианты представлены ниже.

$$\mathbf{F} = \begin{bmatrix} 1 & & x & & 0 \\ & 1 & x & & \\ & & 1 & & \\ & & x & 1 & \\ 0 & & x & & 1 \end{bmatrix} \quad \mathbf{F} = \begin{bmatrix} 1 & & x & & 0 \\ & 1 & x & & \\ & & 1 & & \\ & & & 1 & \\ 0 & & & & 1 \end{bmatrix} \quad \mathbf{F} = \begin{bmatrix} 1 & & & & 0 \\ & 1 & & & \\ & & 1 & & \\ & & x & 1 & \\ 0 & & x & & 1 \end{bmatrix}$$

$$\mathbf{F}=\left[\begin{array}{c|c|c|c|c} 1 & & & & 0 \\ & 1 & & & \\ x & x & 1 & x & x \\ & & & 1 & \\ 0 & & & & 1 \end{array}\right] \quad \mathbf{F}=\left[\begin{array}{c|c|c|c|c} 1 & & & & 0 \\ & 1 & & & \\ x & x & 1 & & \\ & & & 1 & \\ 0 & & & & 1 \end{array}\right] \quad \mathbf{F}=\left[\begin{array}{c|c|c|c|c} 1 & & & & 0 \\ & 1 & & & \\ & & 1 & x & x \\ & & & 1 & \\ 0 & & & & 1 \end{array}\right]$$

Необходимость введения и рассмотрения всех вариантов фробениусовых матриц связана с имеющимся многообразием алгоритмических вариантов прямых методов. Например, столбцовая матрица Фробениуса используется в столбцовом алгоритме метода Гаусса–Жордана, нижняя столбцовая матрица — в столбцовом алгоритме LU-факторизации и т.п. [5].

По-видимому, в иерархию следует включить также класс **MultiRankModificationMatrix**, определяющий группу многоранговых матричных преобразований вида

$$\mathbf{F}=\mathbf{I}+\mathbf{A}\times\mathbf{B}^{T},$$

где $\mathbf{A}$, $\mathbf{B}$ — матрицы соответствующей размерности. Поскольку преобразования данного класса обобщают класс преобразований Фробениуса, класс **FrobeniusMatrix** должен наследоваться от данного. Многоранговые преобразования могут иметь высокую вычислительную сложность, и в этом смысле они не являются столь элементарными. Тем не менее, включение их в общую классификацию связано с той заметной ролью, которую они играют в построении блочных алгоритмов [2].

Важную группу элементарных матриц составляют ортогональные матрицы, объединенные в иерархии абстрактным классом **OrthogonalElementaryMatrix**. Данный матричный класс определяет важное семейство ортогональных преобразований, участвующих в многочисленных методах факторизации, методах решения линейных систем и проблем собственных значений. Конкретные матричные классы **HouseholderMatrix**, **GivensRotationMatrix**, **JacobiRotationMatrix** определяют преобразование Хаусхолдера, преобразования вращения Гивенса и Якоби соответственно.

Матрица преобразования Хаусхолдера **HouseholderMatrix** выражается формулой

$$\mathbf{H}=\mathbf{I}-2\frac{\mathbf{w}\times\mathbf{w}^{T}}{\mathbf{w}^{T}\times\mathbf{w}},$$

где $\mathbf{w}$ — некоторый вектор-столбец. Заметим, что данный класс можно было бы рассматривать и как частный класс матриц Фробениуса. Однако, в силу важности ортогональных преобразований в теории и практике матричного анализа, а также ввиду общности реализации ортогональных методов,

предпочтительнее выглядит используемая классификационная схема.

Наконец, классы **GivensRotationMatrix** и **JacobiRotationMatrix** реализуют преобразования вращения Гивенса и Якоби. Под ними мы будем понимать преобразования, задаваемые следующими матрицами.

$$\mathbf{G} = \begin{bmatrix} 1 & & & & 0 \\ & \cos\Theta_{ij} & & \sin\Theta_{ij} & \\ & & 1 & & \\ & -\sin\Theta_{ij} & & \cos\Theta_{ij} & \\ 0 & & & & 1 \end{bmatrix} \quad \mathbf{J} = \begin{bmatrix} 1 & & & & 0 \\ & 1-2s^2 & & -2cs & \\ & & 1 & & \\ & -2cs & & 1-2c^2 & \\ 0 & & & & 1 \end{bmatrix}$$

Матрицы вращения задают линейное преобразование поворота на некоторый угол Θ_{ij} в плоскости *(ij)*. Матрица Якоби может быть выбрана подобной матрице вращения Гивенса при соответствующем выборе параметров s,c.

Особую группу элементарных матриц составляют основная циркулянтная матрица перестановки класса **BasicCirculant**, а также матрицы сдвига вперед и назад классов **ForwardToeplitz** и **BackwardToeplitz**. Соответствующие им преобразования связаны с решением задач, определяемых над классами циркулянтных и теплицевых матриц.

Таким образом, построена общая классификация элементарных матриц, отражающая их основные математические и вычислительные свойства. Обсудим особенности программной реализации перечисленных классов.

Классы элементарных матриц предназначены, прежде всего, для оперирования с основными матричными объектами класса **GeneralMatrix**, включая и разреженные матрицы, определяемые его подклассами. Разрабатываемые классы элементарных матриц должны при этом обеспечивать необходимую поддержку разреженности. С этой целью используется представление элементарных матриц в виде нескольких (обычно одного или двух) векторов абстрактного типа **Vector**. Поскольку реальные векторные объекты могут поддерживать различные модели разреженности, в том числе и **DenseVector**, **BandVector**, **IndexVector**, **ListVector**, которые описаны в работе [8], удается программно реализовать классы элементарных матриц в самом общем виде без какой-либо конкретизации характера их разреженности.

Представленная выше классификация элементарных матриц определяет минимальный базовый набор матричных преобразований, участвующих в методах линейной алгебры. Возможно, для более экономичной работы с памятью и

эффективной организации вычислений целесообразно расширить описанную классификацию и определить самостоятельные классы, соответствующие целым композициям тех или иных элементарных преобразований. Использование классов такого рода позволило бы избежать затрат на избыточное хранение эквивалентных данных, содержащихся однотипными объектами, и оптимизировать сами процедуры матричных преобразований. В настоящей работе для представления матричных композиций строится специальный контейнерный класс **Composition**, который реализует эту идею в несколько иной форме.

Другая возможность экономии памяти при проведении и хранении необходимых матричных преобразований состоит в размещении их непосредственно на месте основной матрицы. Техника *in-place* размещения часто применяется при процедурной реализации матричных факторизаций. При объектной реализации для этого можно предусмотреть специальные классы элементарных матриц в рамках единой иерархии, которые через ссылку на основную матрицу определяют методы доступа к ее сегментам, хранящим элементы преобразования. В любом случае рассматриваемый подход не исключает и таких возможностей.

3. Объектная реализация классов прямых методов

Обсудим вопросы реализации методов линейной алгебры как объектов соответствующих классов. Для эффективного применения ООП мы нуждаемся, прежде всего, в их целостной объектной классификации. Заметим, что она должна следовать классификации задач линейной алгебры, поскольку результатом решения различных постановок являются объекты разных типов, и это должно отражаться в спецификациях алгоритмических классов. Кроме того, в классификации желательно отразить деление алгоритмов на прямые и итерационные, поскольку принципы их организации существенно отличаются, а это неизбежно приводит к различиям и в их программной реализации.

В силу сделанных замечаний, объектная классификация алгоритмов линейной алгебры может представляться условной иерархией, изображенной на рис. 2.

- **LinearAlgebraAlgorithm**
  - **PivotingAlgorithm**
  - **StructuralAlgorithm**
  - **FactorizationAlgorithm** ←• **(DirectAlgorithm)**
  - **LinearSystemAlgorithm**
    - **DirectLinearSystemAlgorithm**
      - **ExplicitLinearSystemAlgorithm** ←• **(DirectAlgorithm)**
      - **FactorizationLinearSystemAlgorithm**
    - **IterativeLinearSystemAlgorithm** ←• **(IterativeAlgorithm)**
  - **EigenProblemAlgorithm**
  - **LeastSquaresAlgorithm**

Рис. 2. Объектная классификация методов линейной алгебры

В данной схеме все алгоритмы линейной алгебры **LinearAlgebraAlgorithm** классифицируются на вспомогательные алгоритмы поиска главного элемента **PivotingAlgorithm**, необходимые для обеспечения вычислительной устойчивости, структурные алгоритмы **StructuralAlgorithm**, предназначенные для регуляризации портрета матрицы или минимизации его заполнения, алгоритмы факторизации **FactorizationAlgorithm**, а также основные алгоритмы решения СЛАУ **LinearSystemAlgorithm**, проблем собственных значений **EigenProblemAlgorithm** и задач о наименьших квадратах **LeastSquaresAlgorithm**. Перечисленные вспомогательные алгоритмы рассматриваются как самостоятельные классы алгоритмической иерархии, хотя они и не предназначены для решения конечных задач линейной алгебры.

Принципы построения всех алгоритмических классов очень близки. Остановимся более подробно на организации классов алгоритмов решения СЛАУ. Деление алгоритмов данной группы на прямые и итерационные отражено в иерархии введением отдельных классов **DirectLinearSystemAlgorithm**, **IterativeLinearSystemAlgorithm**. Прямые алгоритмы в свою очередь подразделяются на алгоритмы непосредственного решения СЛАУ **ExplicitLinearSystemAlgorithm** и алгоритмы **FactorizationLinearSystemAlgorithm**, использующие предварительную матричную факторизацию того или иного вида.

Анализ прямых и итерационных методов обнаруживает значительную общность в их организации и возможностях унифицированной программной реализации независимо от конкретных типов линейных задач, для решения которых они предназначены. Поэтому имеет смысл построить абстрактные классы прямых и итерационных методов **DirectAlgorithm** и **IterativeAlgorithm** и при реализации конкретных алгоритмов использовать множественное наследование как от соответствующих проблемно-ориентированных классов, так и от

них. Вопросы обобщенной объектной реализации итерационных вычислительных процессов обсуждаются в работе [9] настоящего сборника. Здесь же опишем реализацию класса **DirectAlgorithm**.

Ключевой идеей при программировании прямых методов является представление их в виде композиции элементарных или сложных матричных преобразований

$$\mathbf{F}(\mathbf{A}) = \mathbf{F}_1 \times \mathbf{F}_2 \times ... \mathbf{A} ... \times \mathbf{F}_{k-1} \times \mathbf{F}_k ,$$

которые необходимо применить к исходной задаче для приведения ее к виду, допускающему непосредственное решение или упрощающему его. Конкретизация типов и способов конструирования преобразований, участвующих в композиции, порождает различные классы прямых методов. Совокупность объектов, к которым применяется последовательность преобразований, определяется конкретным классом решаемой задачи. Обычно таким объектом является матрица $\mathbf{A}$, участвующая в постановке линейной задачи.

Предположим, что класс **DirectAlgorithm** определяет чисто виртуальные методы, которые осуществляют:

— определение числа шагов алгоритма *k* по заданным размерностям линейной задачи (или по размерностям характерного матричного объекта),

— определение общего числа преобразований алгоритма и числа преобразований *k* на заданном шаге алгоритма,

— конструирование матрицы преобразования $\mathbf{F}_i, 1 \le i \le k$ по заданной матрице задачи $\mathbf{A}$ на заданном шаге алгоритма,

— определение способа преобразования $\mathbf{F}_i$, которым следует действовать на матрицу $\mathbf{A}$,

— определение блочной подматрицы $\mathbf{A}$, модифицируемой на текущем шаге алгоритма,

— применение вспомогательных алгоритмов структуризации портрета матрицы и поиска главного элемента,

— применение очередного преобразования алгоритма к заданной матрице или вектору,

— применение всего алгоритма.

Тогда любой прямой алгоритм может быть программно оформлен как класс, наследуемый от **DirectAlgorithm**. Для этого в конкретном классе необходимо реализовать перечисленные методы, декларируемые данным классом. Приведем небольшой отрывок из интерфейса класса на языке Си++.

```
/* класс Прямой Алгоритм */
class DirectAlgorithm {
    Matrix* A;                      // указатель на матрицу задачи
    StructuralAlgorithm* structalg; // указатель на алгоритм структуризации
    PivotingAlgorithm* pivotalg;    // указатель на алгоритм выбора главного элемента
```

```
…
public:
…
    virtual Index GetNumberOfStep ( ) = 0;
    virtual Block ActiveSubMatrix (Index step) = 0;
    virtual Block ActiveSubMatrix (Index step, Index j);
    virtual Index GetNumberOfTransform (Index step) = 0;
    virtual Index GetNumberOfTransform ( );
    virtual Matrix* ConstructTransform (Index step, Index j) = 0;
    virtual TSide SideOfTransform (Index step, Index j) = 0;
    virtual void Transform (Index step, Index j, Matrix& A, Matrix& B);
    virtual void Transform (Index step, Index j, Vector& x, Vector& y);
};
```

В данном классе чисто виртуально определена процедура, которая осуществляет конструирование элементарной матрицы, соответствующей преобразованию метода. Кроме того, в данном классе реализуется процедура выполнения очередного шага алгоритма, включающего конструирование факторной матрицы и воздействие ею на заданную матрицу или вектор. Поскольку в методах линейной алгебры могут применяться различные схемы преобразований (слева, справа, с обеих сторон одновременно), в классе определен чисто виртуальный метод, возвращающий направление преобразования перечислимого типа **TSide** = {**TLeft**, **TRight**, **TBothSide**}. Заметим, что данный класс не осуществляет хранение факторных матриц и всей композиции, а только определяет методы их конструирования. В самом деле, придание классу таких функций оказалось бы совершенно излишним, например, при реализации алгоритмов гауссовского исключения, в ходе которого не требуется запоминать преобразования, примененные к матрице и правой части СЛАУ.

Для представления композиций матричных преобразований используется вспомогательный контейнерный класс **Composition**, осуществляющий их динамическое размещение.

```
/* класс Матричная Композиция */
class Composition {
    Index number;        // число факторных матриц
    Matrix** pmatrix;    // массив факторных матриц
    TSide* side;         // массив направлений преобразований
    Bool* degree;        // массив степеней преобразований
…
public:
…
    Index GetNumberOfFactor ( );
    void SetNumberOfFactor (Index n);
    void SetFactor (Index i, Matrix& F, TSide side, Bool degree);
    Matrix& GetFactor (Index i);
    TSide GetSide (Index i);
    Bool GetDegree (Index i);
    void ApplyTo (Index i, Matrix& A);
    void ApplyTo (Index i, Vector& x);
};
```

Методы класса **Composition** реализуют доступ к факторным матрицам, осуществляют их включение в композицию, а также реализуют процедуру воздействия ими на заданную матрицу или вектор. Для каждой факторной матрицы, включенной в композицию, хранится способ действия типа **TSide** и ее степень, которая принимает одно из двух значений 1 или –1 и представляется типом Bool. Необходимость представления степени связана с тем, что часто преобразование задается не самой матрицей, а обратной к ней. Явное обращение матрицы может приводить к большой вычислительной погрешности и ухудшению разреженности. Использование степени оказывается удобным, поскольку для представления преобразования достаточно иметь средства, позволяющие применять его к заданным матричным или векторным объектам. В случае положительной степени действие преобразования сводится к умножению факторной матрицы на заданную, а в случае отрицательной — к решению соответствующей СЛАУ, что является более предпочтительным, чем явное обращение матрицы.

Другой особенностью данного класса является организация матричной композиции как массива матриц произвольного типа **Matrix**. Подобная общность связана с тем, что результатом применения алгоритмов факторизации и других матричных алгоритмов, имеющих целью получение композиционного представления того или иного вида, могут быть не только элементарные, но и произвольные матрицы. Например, в методах LU-, QR-факторизации применяется декомпозиция матрицы на треугольные сомножители, в методах нахождения собственных значений строится композиция преобразований, приводящая матрицу к хессенберговой форме и т.п.

По-видимому, классы **DirectAlgorithm** и **Composition** максимально унифицируют программные средства, необходимые для конструирования и представления матричных преобразований в прямых методах линейной алгебры. С их использованием удается реализовать наиболее значимые классы алгоритмов линейной алгебры, включая алгоритмы факторизации **FactorizationAlgorithm**.

В самом деле, любой алгоритм факторизации имеет целью получение декомпозиционного представления заданной матрицы

$$\mathbf{A} = \mathbf{F}_1 \times ... \times \mathbf{F}_i ,$$

где $\mathbf{A}$ — исходная матрица, $\mathbf{F}_i$ — *i*-ая матрица факторизации. Выделение методов факторизации в самостоятельную группу связано со спецификой построения факторных матриц, которые теперь интерпретируются не как матрицы преобразования, а как

матрицы декомпозиции. Все методы родительского класса **DirectAlgorithm** при этом автоматически наследуются, сохраняя близкую семантическую нагрузку. Вид матриц $\mathbf{F}_i$ и их число определяются конкретным алгоритмом. Данный класс также формально наследуется от класса **LinearAlgebraAlgorithm**, что необходимо для поддержания стилистического единства библиотеки.

Приведем пример программной реализации класса **FactorizationAlgorithm**.

```
/* класс Алгоритм Факторизации */
class FactorizationAlgorithm: public LinearAlgebraAlgorithm, DirectAlgorithm {
    Composition* comp;    // указатель на матричную композицию
…
public:
…
    virtual Matrix* ConstructRemainder ( ) = 0;
    void Run (Matrix& matrix, Composition& composition);
};
```

В классе дополнительно определены метод проведения факторизации и метод конструирования части декомпозиции, остающейся после применения соответствующих преобразований. Важно отметить, что репертуар методов абстрактного класса **FactorizationAlgorithm** позволяет представить некоторый обобщенный алгоритм факторизации и реализовать его непосредственно в данном классе один раз. Построение конкретных алгоритмов в этом случае осуществляется путем создания частных наследуемых классов и реализации чисто виртуальных функций, определяющих отдельные компоненты обобщенного алгоритма.

Рассмотрим возможную программную реализацию данного алгоритма, следуя фрагменту основного метода класса **FactorizationAlgorithm**, выполняющего факторизацию.

```
void FactorizationAlgorithm::Run(Matrix& matrix, Composition &composition) {
    A=&matrix;
    comp=&composition;
//  устанавливаем число факторных матриц в композиции
    comp–>SetNumberOfFactor(GetNumberOfTransform( )+3);
//  заводим матрицы перестановок строк и столбцов
    PermutationMatrix *prow = new PermutationMatrix(A–>NSizeOf( ));
    PermutationMatrix *pcol = new PermutationMatrix(A–>MSizeOf( ));
//  применяем алгоритм структуризации портрета матрицы
    structalg–>Run(*A, *prow, *pcol);
//  проводим необходимое переупорядочивание
    *A>>=*prow;
    *A<<=*pcol;
//  заводим элементарные матрицы перестановок строк и столбцов
    TranspositionMatrix *pelmrow = new PermutationMatrix(A–>NSizeOf( ));
    TranspositionMatrix *pelmcol = new PermutationMatrix(A–>MSizeOf( ));
    Index k=1;
//  цикл по шагам алгоритма
    for(Index i=0; i<GetNumberOfStep( ); i++) {
```

```
// применяем алгоритм поиска главного элемента
      pivotalg–>Run(*A, ActiveSubMatrix(i), *pelmrow, *pelmcol);
// проводим необходимую перестановку строк и столбцов
      *A>>=*pelmrow;
      *A<<=*pelmcol;
// корректируем матрицы перестановок
      *prow>>=*pelmrow;
      *pcol<<=*pelmcol;
// цикл по числу преобразований на текущем шаге алгоритма
      for(Index j=0; j<GetNumberOfTransform(i); j++, k++) {
// конструируем преобразование и включаем его в композицию
         comp–>SetFactor(k, ConstructTranform(i, j), SideOfTransform(i, j), False);
// применяем преобразование
         comp–>ApplyTo(k, *A, ActiveSubMatrix(i, j));
      }
   }
// включаем матрицы перестановок и оставшуюся часть матрицы в композицию
   comp–>SetFactor(0, prow, TLeft, False);
   comp–>SetFactor(k, ConstructRemainder( ), TLeft, True);
   comp–>SetFactor(k+1, pcol, TRight, False);
}
```

Нетрудно видеть, что организация классов **DirectAlgorithm**, **FactorizationAlgorithm** удачно сочетается с необходимостью применения вспомогательных методов структуризации портрета матрицы **StructuralAlgorithm** перед началом факторизации и методов поиска главного элемента **PivotingAlgorithm** в ходе ее проведения. Привлечение данных методик позволяет обеспечить вычислительную эффективность в довольно сложных ситуациях и является крайне важным при работе с разреженными и плохообусловленными матрицами. Фактически, описанный класс **FactorizationAlgorithm** представляет собой мощное инструментальное средство для конструирования комбинированных алгоритмов, включающих в себя необходимые алгоритмы факторизации, структуризации и выбора главного элемента. С учетом того, что каждый вспомогательный алгоритм может быть выбран самым произвольным образом, данная программная реализация обеспечивает значительную алгоритмическую общность.

Другой существенной стороной данной реализации является полная унификация методов оперирования с элементарными и сложными матричными преобразованиями, необходимыми для проведения факторизации. Поскольку в классах **FactorizationAlgorithm**, **Composition** не конкретизируются типы матричных преобразований (типы факторных матриц), удается реализовать самую общую схему матричной факторизации, в которой вид, а также способы конструирования, применения и хранения преобразований определяются конкретным алгоритмическим классом.

Более того, на основе данных классов легко построить классы обобщенных методов решения СЛАУ и спектральных задач. Рассмотрим, например, программную реализацию класса **FactorizationLinearSystemAlgorithm**, обобщающего методы решения СЛАУ с использованием предварительной факторизации.

```
/* класс Алгоритм Решения Линейных Систем на основе Факторизации */
class FactorizationLinearSystemAlgorithm: public LinearSystemAlgorithm {
    Matrix* A;                          // указатель на матрицу системы
    Vector* y;                          // указатель на вектор правой части
    Vector* x;                          // указатель на вектор решения
    FactorizationAlgorithm* factoralg;  // указатель на алгоритм факторизации
    Composition* comp;                  // указатель на декомпозицию
…
};
void FactorizationLinearSystemAlgorithm::Run(Matrix& matrix, Vector& xvec, Vector& yvec) {
    A=&matrix;
    x=&xvec;
    y=&yvec;
//  заводим матричную композицию
    comp= new Composition;
//  ставим задачу факторизации и применяем алгоритм
    factoralg–>Run(*A, *comp);
//  разрешаем систему относительно каждой факторной матрицы, входящей в декомпозицию
    x=y;
    for(Index i=0; i<comp–>GetNumberOfFactor( ); i++)
        if (comp–>GetDegree(i))
            comp–>GetFactor(i).SolveLinearSystem(x, x);
        else
            comp–>GetFactor(i).MultiplyBy(x, x);
}
```

Заметим, что в данной программе используется абстрактный тип алгоритма факторизации. Единственным требованием, неявно предъявляемым к нему, является условие декомпозиции матрицы на элементарные множители, для которых определены методы решения СЛАУ и операции матричного умножения. Последние необходимы в случае задания декомпозиционного представления с помощью обратных матриц.

Таким образом, разработанный ООП к программированию прямых методов линейной алгебры обеспечивает значительную алгоритмическую общность, а разработанные алгоритмические классы могут использоваться как в качестве стандартных программных средств постановки и решения линейных задач, так и в качестве инструментария для их дальнейшего развития. Программирование методов факторизации сводится к построению частных алгоритмических классов, наследуемых от абстрактного класса **FactorizationAlgorithm**, и реализации методов конструирования матричных преобразований, участвующих в конкретных видах факторизации.

Перейдем к рассмотрению конкретных классов алгоритмов факторизации.

4. Объектная классификация методов факторизации

Следуя развитому ООП, методы матричной факторизации будем рассматривать как объекты целостной классификационной иерархии, порождаемой алгоритмическим суперклассом **FactorizationAlgorithm**. Ниже приведена возможная объектная классификация методов факторизации, представляющая наиболее важные группы методов и отражающая методологическую общность в подходах, реализуемых ими (рис. 3). Из-за краткости в нее не были включены ветви алгоритмов, аналогичные уже имеющимся. В основе данной классификации лежит принцип анализа типов матриц (элементарных или общих), к которым приводит тот или

- **FactorizationAlgorithm**
  - **LUAlgorithm**
    - **LURowUpAlgorithm**
    - **LURowDownAlgorithm**
    - **LUColumnUpAlgorithm**
    - **LUColumnDownAlgorithm**
  - **LDUAlgorithm**
  - **GaussJordanAlgorithm**
  - **OrthogonalFactorizationAlgorithm**
    - **QRFactorizationAlgorithm**
      - **HouseholderQRAlgorithm**
      - **GivensQRAlgorithm**
    - **LQFactorizationAlgorithm**
  - **SingularAlgorithm**
  - **PolarAlgorithm**
  - **SymmetricFactorizationAlgorithm**
    - **LLTCholeskyAlgorithm**
    - **LDLTCholeskyAlgorithm**
    - **OrthogonalSymmetricFactorizationAlgorithm**
      - **JacobiQRAlgorithm**
    - **TakagiAlgorithm**

Рис. 3. Объектная классификация методов матричной факторизации

иной метод факторизации.

Вершиной данной иерархии является класс **FactorizationAlgorithm**, представляющий некоторый обобщенный алгоритм факторизации, а также определяющий и частично реализующий его отдельные компоненты. Следующий уровень иерархии занимают классы **LUAlgorithm**, **LDUAlgorithm**, **GaussJordanAlgorithm**, **OrthogonalFactorizationAlgorithm** и **SymmetricFactorizationAlgorithm**, каждый из которых обобщает группы методов, соответствующих определенным типам разложений.

Абстрактный класс **LUAlgorithm** определяет группу методов, приводящих к различным формам LU-разложения

$$\mathbf{A} = \mathbf{L} \times \mathbf{U},$$

в котором $\mathbf{L}$ — нижне-, а $\mathbf{U}$ — верхнетреугольная матрица. Заметим, что результатом применения методов данного класса могут быть и композиции элементарных матриц, соответствующие отдельным треугольным факторам. Вообще говоря, подобные представления отличаются от классического варианта LU-разложения. Существенным является лишь то, что способ композиции элементарных матриц в конечном итоге определяет именно такую форму разложения. Рассмотрим конкретные классы методов LU-разложения, наследуемые от данного.

Класс **LUColumnDownAlgorithm** определяет наиболее распространенную модификацию LU-разложения. Суть метода состоит в последовательном обнулении ненулевых элементов под главной диагональю по столбцам. С математической точки зрения метод заключается в последовательном умножении исходной матрицы на элементарную нижне-столбцовую матрицу Фробениуса **FrobeniusColumnDownMatrix** справа, в результате чего формируется верхнетреугольная матрица. В матричной записи *k*-ый шаг алгоритма имеет вид

$$\mathbf{A}_{k+1} = \mathbf{F}_k \times \mathbf{A}_k,$$

где $\mathbf{A}_k$ — факторизуемая матрица, $\mathbf{F}_k$ — матрица факторизации. Элементы ненулевого *k*-ого столбца матрицы факторизации $\mathbf{f}_k$ конструируются следующим образом:

$$\mathbf{f}_k = \left[0,...,0,1,-\frac{a_{k+1,k}}{a_{k,k}},...,-\frac{a_{n-1,k}}{a_{k,k}}\right]^T.$$

Выражение для факторизованной матрицы приобретает окончательный вид

$$\mathbf{A} = \left(\mathbf{F}_{n-2} \times ... \times \mathbf{F}_0\right)^{-1} \times \mathbf{U} = \mathbf{L} \times \mathbf{U}.$$

Обсудим программную реализацию методов факторизации на примере класса **LUColumnDownAlgorithm**. В конкретном классе обязаны быть определены все методы, объявленные чисто виртуальными в родительских классах. Перечислим методы базовых алгоритмических классов **DirectAlgorithm**, **FactorizationAlgorithm**, которые следует реализовать в классе **LUColumnDownAlgorithm**.

Прежде всего, метод *GetNumberOfStep( )*, определяющий число шагов алгоритма, должен возвращать значение *n*–1, где *n* — размерность факторизуемой квадратной матрицы. На каждом шаге алгоритма применяется одно преобразование, и метод

GetNumberOfTransform(Index step) возвращает 1. В качестве матриц факторизации используются нижние столбцовые матрицы Фробениуса, которые применяются слева, поэтому метод *SideOfTransform(Index step, Index j)* возвращает значение *TLeft*, а метод *ConstructTransform(Index step, Index j)* конструирует необходимую элементарную матрицу преобразования в соответствии с приведенным правилом и возвращает указатель на нее.

```
Matrix* LUColumnDownAlgorithm::ConstructTransform (Index step, Index j) {
    FrobeniusColumnDownMatrix *f=new FrobeniusColumnDownMatrix(A,step);
    return f;
}
```

В данном методе, как и в большинстве других методов факторизации, число оставшихся шагов совпадает с порядком активной модифицируемой подматрицы, поэтому процедурой *ActiveSubMatrix(Index step)* возвращается блок, соответствующий шагу алгоритма. Наконец, метод *ConstructRemainder( )* конструирует верхнетреугольную матрицу, к которой была приведена исходная матрица в результате применения всех преобразований.

Набор данных методов является достаточным для факторизации произвольной матрицы с использованием обобщенной методики *Run(Matrix& matrix, Composition& composition)* класса **FactorizationAlgorithm**. Основным моментом описанной унифицированной реализации методов факторизации является построение элементарных преобразований, поэтому в дальнейшем ограничимся именно этим аспектом.

Класс **LURowDownAlgorithm** — алгоритмический вариант LU-разложения по строкам. Суть метода заключается в последовательном обнулении ненулевых элементов под главной диагональю по строкам, начиная с последней. Данный метод имеет два основных отличия от предыдущего. Во-первых, изменен порядок факторизации исходной матрицы

$$\mathbf{A}_{k+1} = \mathbf{A}_k \times \mathbf{F}_k .$$

Во-вторых, в качестве матрицы факторизации на *k*-ом шаге применяется нижне-строчная матрица Фробениуса $\mathbf{F}_k$ класса **FrobeniusRowDownMatrix** с ненулевой (*n–k–1*)-ой строкой

$$\mathbf{f}_{n-k-1} = \left[-\frac{a_{n-k-1,0}}{a_{n-k,n-k}}, \ldots, -\frac{a_{n-k-1,n-k-2}}{a_{n-k,n-k}}, 1, 0, \ldots, 0 \right].$$

Факторизованная матрица приобретает окончательный вид

$$\mathbf{A} = \mathbf{U} \times \left(\mathbf{F}_0 \times \ldots \times \mathbf{F}_{n-2} \right)^{-1} = \mathbf{U} \times \mathbf{L} .$$

Класс **LUColumnUpAlgorithm** определяет вариант столбцового LU-разложения, в котором на каждом *k*-ом шаге используется

верхняя столбцовая матрица Фробениуса $\mathbf{F}_k$ класса **FrobeniusColumnUpMatrix** с ненулевым (*n*–*k*+1)-ым столбцом

$$\mathbf{f}_{n-k-1} = \left[-\frac{a_{0,n-k-1}}{a_{n-k,n-k}},...,-\frac{a_{n-k-2,n-k-1}}{a_{n-k,n-k}},1,0,...,0 \right]^T .$$

Порядок данной факторизации левосторонний, конечный вид разложения выражается формулой

$$\mathbf{A} = \left(\mathbf{F}_0 \times ... \times \mathbf{F}_{n-2}\right)^{-1} \times \mathbf{L} = \mathbf{U} \times \mathbf{L} .$$

Класс **LURowUpAlgorithm** реализует вариант строчного LU-разложения, в котором на *k*-ом шаге в качестве матрицы факторизации применяется верхняя строчная матрица Фробениуса **FrobeniusRowUpMatrix** с ненулевой *k*-ой строкой

$$\mathbf{f}_k = \left[0,...,0,1,-\frac{a_{k,k+1}}{a_{k,k}},...,-\frac{a_{k,n-1}}{a_{k,k}} \right].$$

Порядок данной факторизации правосторонний, конечный вид разложения определяется формулой

$$\mathbf{A} = \mathbf{L} \times \left(\mathbf{F}_0 \times ... \times \mathbf{F}_{n-2}\right)^{-1} = \mathbf{L} \times \mathbf{U} .$$

Алгоритмы класса **LDUAlgorithm** реализуют варианты LU-разложения, в которых различные виды матриц Фробениуса чередуются с масштабирующими матрицами типа **ScalingMatrix**. В качестве примера конкретных методов данного класса рассмотрим алгоритм нижней столбцовой факторизации **LDUColumnDownAlgorithm**. Данный метод проводит разложение вида

$$\mathbf{A} = (\mathbf{D}_{n-1} \times \mathbf{F}_{n-2} \times \mathbf{D}_{n-2} \times ... \times \mathbf{F}_0 \times \mathbf{D}_0)^{-1} \times \mathbf{U} .$$

На каждом *k*-ом шаге метода применяется два преобразования $\mathbf{F}_k$, $\mathbf{D}_k$, соответствующие нижне-столбцовой матрице Фробениуса **FrobeniusColumnDownMatrix** с ненулевым *k*-ым столбцом вида

$$\mathbf{f}_k = \left[0,...,0,1,-a_{k+1,k},...,-a_{n-1,k} \right]^T$$

и масштабирующей матрице класса **ScalingMatrix** с главным элементом $d_{k,k} = 1 \big/ a_{k,k}$. В результате применения всех преобразований получается верхнетреугольная матрица с единичной главной диагональю $\mathbf{U}$.

В класс **LDUAlgorithm** могут быть включены также алгоритмы **LDURowUpAlgorithm**, **LDURowDownAlgorithm** и **LDUColumnUpAlgorithm**, различающиеся типами применяемых преобразований Фробениуса и способами действия ими. Данная группа алгоритмов полностью аналогична рассмотренной выше группе **LUAlgorithm**.

Алгоритмы, представленные в иерархии классом **GaussJordanAlgorithm**, реализуют различные варианты метода Гаусса–Жордана. В ходе проведения данного разложения одновременно исключаются все внедиагональные элементы текущей строки или столбца. Итогом преобразований является диагональная матрица. Наиболее широко применяется столбцовый вариант данного алгоритма — **GaussJordanColumnAlgorithm**. Факторизация матрицы в данном методе производится слева с использованием элементарной столбцовой матрицы Фробениуса класса **FrobeniusColumnMatrix**. Окончательная форма разложения имеет вид

$$\mathbf{A} = \left(\mathbf{F}_{n-2} \times ... \times \mathbf{F}_0\right)^{-1} \times \mathbf{D} ,$$

в котором матрица Фробениуса $\mathbf{F}_k$ определяется ненулевым столбцом

$$\mathbf{f}_k = \left[-\frac{a_{0,k}}{a_{k,k}}, ..., -\frac{a_{k-1,k}}{a_{k,k}}, 1, -\frac{a_{k+1,k}}{a_{k,k}}, ..., -\frac{a_{n-1,k}}{a_{k,k}} \right]^T ,$$

а $\mathbf{D}$ является результирующей диагональной матрицей.

Класс **OrthogonalFactorizationAlgorithm** обобщает методы факторизации, использующие ортогональные преобразования. Методы данного класса широко применяются как при решении спектральных задач, так и при решении СЛАУ. В силу особенностей ортогональных преобразований, связанных с сохранением спектра матрицы и простотой обращения самих ортогональных матриц, данные методы выделены в отдельную классификационную ветвь. В ней представлены две группы алгоритмов **QRFactorizationAlgorithm** и **LQFactorizationAlgorithm**, реализующие QR- и LQ-факторизацию.

Класс **QRFactorizationAlgorithm** обобщает группу ортогональных методов, приводящих к декомпозиции матрицы на ортогональную $\mathbf{Q}$ и верхнетреугольную $\mathbf{R}$

$$\mathbf{A} = \mathbf{Q} \times \mathbf{R} .$$

Класс **QRHouseholderAlgorithm** представляет конкретный алгоритм QR-факторизации с использованием преобразования Хаусхолдера. Данный метод заключается в последовательном обнулении столбцов под главной диагональю путем умножения ее слева на элементарную матрицу Хаусхолдера типа **HouseholderMatrix**. Данный алгоритм в матричных терминах имеет вид

$$\mathbf{A} = \left(\mathbf{H}_{n-2} \times ... \times \mathbf{H}_0\right)^{-1} \times \mathbf{R} ,$$

где элементарная матрица Хаусхолдера $\mathbf{H}_k$ на *k*-ом шаге строится следующим образом:

$$\mathbf{H}_k = \mathbf{I} - \mathbf{w}_k \times \mathbf{w}_k^T,$$

$$\mathbf{w}_k = \mu_k * \mathbf{u}_k,\ \mathbf{u}_k = \left[0,...,0,a_{k,k} - s_k, a_{k+1,k},...,a_{n-1,k}\right]^T,$$

$$s_k = -\operatorname{sgn}(a_{k,k}) * \sqrt{\sum_{i=k}^{n-1} a_{i,k}^2},\ \mu_k = 1 \Big/ \sqrt{s_k^2 - a_{k,k} * s_k}.$$

Класс **QRGivensAlgorithm** реализует ортогональный метод QR-разложения, использующий вращения Гивенса. Данный метод осуществляет поэлементное исключение в поддиагональной части матрицы, поэтому на каждом шаге применяется целая серия преобразований Гивенса класса **GivensRotationMatrix**. Метод использует всего $n*(n-1)/2$ преобразований и приводит к верхнетреугольной матрице

$$\mathbf{A} = \left(\mathbf{G}_{n-1,n-2} \times \mathbf{G}_{n-1,n-3} \times \mathbf{G}_{n-2,n-3} \times ... \times \mathbf{G}_{2,1} \times \mathbf{G}_{n-1,0} \times ... \times \mathbf{G}_{1,0}\right)^{-1} \times \mathbf{R},$$

где матрица Гивенса $\mathbf{G}_{i,j}$ выбирается следующим образом:

$$\cos\Theta = a_{i,i} \Big/ \gamma,\ \sin\Theta = a_{j,j} \Big/ \gamma,\ \gamma = \sqrt{a_{i,i}^2 + a_{i,j}^2}.$$

Близкий класс **QRJacobiAlgorithm** осуществляет QR-факторизацию симметричной матрицы с помощью преобразований вращения Якоби класса **JacobiRotationMatrix**.

Подобно классам **QRFactorizationAlgorithm** может быть организовано семейство классов **LQFactorizationAlgorithm**, включающее ортогональные методы декомпозиции матрицы на нижнетреугольную и ортогональную матрицы

$$\mathbf{A} = \mathbf{L} \times \mathbf{Q}.$$

Важный класс алгоритмов **SymmetricFactorizationAlgorithm** составляют алгоритмы факторизации симметричных матриц. Выделение в отдельную классификационную ветвь связано со свойством симметрии, ограничивающим их применение в случае произвольных матриц. Заметим, что для факторизации симметричных матричных объектов могут использоваться и приведенные выше универсальные алгоритмы. Для этого достаточно преобразовать симметричный матричный объект к одному из общих матричных типов класса **Matrix**.

Класс **LLTCholeskyAlgorithm** реализует разложение Холецкого. В ходе проведения разложения на каждом *k*-ом шаге исключаются элементы *k*-ой строки и *k*-ого столбца симметричной матрицы. Для этого исходная матрица умножается на матрицу Фробениуса слева и справа. В результате преобразований матрица приводится к единичной. Математически данная факторизация записывается в виде

$$\mathbf{F}_{n-1} \times ... \times \mathbf{F}_0 \times \mathbf{A} \times \mathbf{F}_0^T \times ... \times \mathbf{F}_{n-1}^T = \mathbf{I},$$

$$\mathbf{A} = \left(\mathbf{F}_{n-1} \times ... \times \mathbf{F}_0\right)^{-1} \times \mathbf{I} \times \left(\left(\mathbf{F}_{n-1} \times ... \times \mathbf{F}_0\right)^{-1}\right)^T = \mathbf{L} \times \mathbf{L}^T ,$$

где $\mathbf{F}_k$ — нижняя столбцовая матрица Фробениуса **FrobeniusColumnDownMatrix** с ненулевым *k*-ым столбцом

$$\mathbf{f}_k = \left[0,...,0,f_{k,k},...,f_{n-1,k}\right]^T ,$$

$$f_{k,k} = \sqrt{a_{k,k} - \sum_{i=0}^{k-1} f_{k,i}^2} \ , \ f_{i,k} = \left(a_{i,k} - \sum_{j=0}^{k-1} f_{i,j} \times f_{k,j}\right) \Big/ f_{k,k} .$$

Основное отличие данного метода от предыдущих заключается в применении двухсторонних преобразований к симметричной матрице. Класс **LDLTCholeskyAlgorithm** представляет одну из модификаций разложения Холецкого

$$\mathbf{A} = \mathbf{L} \times \mathbf{D} \times \mathbf{L}^T ,$$

в котором результирующая матрица $\mathbf{D}$ диагональна, а нижнетреугольная матрица $\mathbf{L}$ имеет единичную диагональ.

Приведенные выше алгоритмы составляют классы наиболее популярных методов факторизации, нашедших широкое применение в многочисленных приложениях. Тщательное перечисление методов факторизации имело целью показать возможности унифицированного математического представления их в виде композиций элементарных матричных преобразований и доказать общность предложенного ООП к программированию прямых методов, опирающегося на данное представление. В объектную классификацию могли бы быть включены и многие другие методы, в частности, методы факторизации комплекснозначных матриц [4]. Возможность расширения классификации формально обозначена введением дополнительных классов алгоритмов сингулярного разложения **SingularAlgorithm**, разложения Такаги **TakagiAlgorithm**, полярного разложения **PolarAlgorithm**.

5. Заключение

Таким образом, предложен ООП к программированию прямых методов линейной алгебры с использованием технологии элементарных матриц. Подход опирается на единое математическое представление прямых методов в виде композиций элементарных и сложных матричных преобразований и заключается в обобщенной реализации методов как объектов единой алгоритмической классификации с использованием известных объектных парадигм. Предлагаемый подход обеспечивает значительную алгоритмическую общность и возможность унифицированной реализации широких классов

алгоритмов линейной алгебры, включая сложные комбинированные алгоритмические схемы с использованием методов поиска главного элемента, методов структуризации и минимизации заполнения.

Подход апробирован при практической реализации математической библиотеки [7] на языке Си++. Разработанные матричные, алгоритмические и проблемные классы библиотеки могут использоваться в качестве программных средств постановки и решения стандартных задач линейной алгебры, а также в качестве мощного инструментария для их дальнейшего расширения, модификации и адаптации к конкретным прикладным проблемам.

Работа поддержана Российским Фондом фундаментальных исследований (грант 95–01–01239).

ЛИТЕРАТУРА

1. Воеводин В.В. Линейная алгебра. — М.: Наука, 1980.
2. Тыртышников Е.Е. Блочные алгоритмы линейной алгебры. // Вычислительные процессы и системы, вып. 9. — М.: Наука, 1993, с. 3–31.
3. Гантмахер Ф.Р. Теория матриц. — М.: Наука, 1988.
4. Хорн Р., Джонсон Ч. Матричный анализ: Пер. с англ. — М.: Мир, 1989.
5. Писсанецки С. Технология разреженных матриц: Пер. с англ. — М.: Мир, 1988.
6. Proceedings of the Second Annual Object-Oriented Numerics Conference. Sunriver, Oregon, 1994.
7. Семенов В.А. Об объектно-ориентированном подходе к разработке численного математического обеспечения. // Вопросы кибернетики. Приложения системного программирования. — М.: НСК РАН, 1995, с. 140–163.
8. Семенов В.А., Тарлапан О.А. Технологии реализации разреженных матричных классов. // Вопросы кибернетики. Приложения системного программирования. — М.: НСК РАН, 1995, с. 164–188.
9. Семенов В.А., Морозов С.В. Объектно-ориентированное программирование квадратурных методов. // Настоящий сборник.