Лекция 3

Общие сведения о возможности использования метода Жордана-Гаусса для решения СЛАУ

- Норма вектора

  Пусть

  $$x = [x_1, \dots, x_N] \in \mathbb{R}^N (или\ \mathbb{C}^N)$$

  Евклидова норма (норма 2-го порядка) N-мерного вектра определяется как

  $$\|x\|_2 = \sqrt{|x_1|^2 + \dots + |x_N|^2}$$

  Часто используется также норма 1-го порядка:

  $$\|x\|_1 = |x_1| + \dots + |x_N|$$

  а так же равномерная норма (норма бесконечного порядка):

  $$\|x\|_\infty = \max\left(|x_1| + \dots + |x_N|\right)$$

  Вообще говоря, величина нормы вектора характеризует степень отклонения вектрора от нулевого вектора.
  При этом норма разности каких-либо двух радиус-вектров характеризует величину отклонения соответсвующих им точек N-мерного пространства друг от друга.

  Существуют и другие нормы вектров. Любая норма подчиняется следующим трем аксиомам:

  - $\|x\| \le 0$, причем $\|x\| = 0 \Leftrightarrow x = 0$
  - $\|cx\| = |c|\|x\|\ \forall c \in \mathbb{R}\ (или\ \mathbb{C})$
  - $\|x - y\| \le \|x - z\| + \|y - z\|\ \forall x, y, z \in \mathbb{R}^N\ (или\ \mathbb{C}^N)$ - это свойство выражает так называемое неравенство треугольника.

- Норма матрица, согласованная с данной векторной нормой

  $$\|A\|_p = \max_{x \in \mathbb{R}^N} \frac{\|Ax\|_p}{\|x\|_p}$$

Очевидно следующее важное неравенство

$$\|Ax\|_p \leq \|A\|_p \|x\|_p$$

- Число обусловленности квадратной невырожденной матрицы

$$\mathrm{cond}(A) = \max_{y \in \mathbb{R}^N} \frac{\|A^{-1}y\|_p}{\|y\|_p} \max_{x \in \mathbb{R}^N} \frac{\|Ax\|_p}{\|x\|_p} = \|A^{-1}\|_p \|A\|_p$$

Эта характеристика матрицы характеризует степень близости матрицы к вырожденной матрице. Чем больше число обусловленности, тем ближе в этом смысле матрица к вырожденной. Для любой вырожденной матрицы, число обусловленности считается равным $+\infty$.

В пакте LinearAlgebra языка Julia имеется специальная функция, вычисляющая норму матрицы порядка p:

```
cond(M, p::Real=2)
```

(по умолчанию `p=2`).

- Оценка погрешности решения СЛАУ

Пусть

$$Ax = b$$

И пусть при этом правая часть этой системы задана с некоторой абсолютной погрешностью Δb. Тогда решение рассматриваемой системы будет отличаться от точного решения, и будет иметь абсолютную погрешность Δx. Т.е. если

$$A\tilde{x} = \tilde{b}$$

где
$\tilde{b} = b + \Delta b$, то $\tilde{x} = x + \Delta x$.

Вычитая из второго равенства первое, очевидно, получаем

$$A\Delta x = \Delta b$$

или

$$\Delta x = A^{-1}\Delta b$$

Откуда

$$\|\Delta x\|_p = \|A^{-1}\Delta b\| \leq \|A^{-1}\|_p \|\Delta b\|_p$$

Далее, разделив обе части последнего неравенства на $\|x\|_p$, получим:

$$\frac{\|\Delta x\|_p}{\|x\|_p} \leq \|A^{-1}\|_p \frac{\|\Delta b\|_p}{\|x\|_p} = \|A^{-1}\|_p \frac{\|\Delta b\|_p}{\|x\|_p} == \|A^{-1}\|_p \|A\|_p \frac{\|\Delta b\|_p}{\|A\|_p \|x\|_p} \leq \|A^{-1}\|_p \|A\|_p \frac{\|\Delta b\|_p}{\|Ax\|_p} = \|A^{-1}\|_p \|.$$

Откуда, вводя обозначения для относительных погрешностей
$\delta x = \frac{\|\Delta x\|_p}{\|x\|_p}$ и $\delta b = \frac{\|\Delta b\|_p}{\|b\|_p}$
при условии $\|b\| \neq 0$, $\|x\| \neq 0$, получаем искомую оценку сверху

$$\delta x \leq \mathrm{cond(A)} \delta b$$

Отсюда вида видно, что при очень больших значениях числа обусловленности $\mathrm{cond(A)}$ возможно катастрофическое увеличение относительной погрешности решения, даже при условии малости относительной погрешности правой части СЛАУ.

Если погрешность имеется в коэффициентах матрицы A, т.е.

$$\tilde{A} = A + \Delta A = (I - \Delta A A^{-1}) A$$

то систему

$$\tilde{A} x = b$$

можно переписать в виде

$$Ax = (I - \Delta A A^{-1})^{-1} b = \tilde{b}$$

где $\tilde{b} = (I - \Delta A A^{-1}) b$ (здесь предполагается, что матрица $\Delta A A^{-1}$ имеет малую величину, т.е. малую норму).

Таким образом, анализ влияния погрешностей в значениях коэффициентов матрицы также может быть сведен к анализу погрешностей в значениях вектора правой части СЛАУ.

Если число обусловленности матрицы исчисляется тысячами, то такая ситуация, при вычислениях в формате с плавающей точкой `Float64`, можно считать приемлемой для применения метода Жордана-Гаусса. В противном случае погрешности вычислений могут достигать неприемлемо высоких значений.

- Пример плохообусловленной СЛАУ

$$\begin{cases} x_1 + 10 x_2 = 11 \\ 100 x1 + 1001 x_2 = 1101 \end{cases}$$

Решение: $x_1 = 1, x_2 = 1$

$$\begin{cases} x_1 + 10 x_2 = 11.01 \\ 100 x1 + 1001 x_2 = 1101 \end{cases}$$

Решение: $x_1 = 11.01; x_2 = 0.0$

Ответ изменился на порядок, тогда как правая часть уравнения измениласть всего лишь на 0.1%!

Это объясняется тем, что мартица этой сисемы плохо обусловлена, число обусловленности имеет порядок числа 1_000_000

```
julia> cond([1     10
             100 1001])

1.012102000015063e6
```

Графическая интерпретация рассморенного примера состоит в том, что прямые, которые задаются каждым из уравнений рассмотренной системы, оказываются почти параллельными. Из-за этого и возникает пробле точного определения координат точки пересечения этих прямых, т.к. малейшее "шевеление" графиков этих прямых будет приводить к очень большим изменениям положения точки пересечения.

Программная реализация алгоритма Жордана-Гасса

Обратный ход метода Жордана-Гаусса

Как известно, решение СЛАУ методом Жордана-Гаусса состоит из двух частей: сначала расширенная матрица системы с помощью элементарных преобразований её строк приводится к ступенчатому виду, а затем по очевидной простой формуле вычисляются значения элементов вектора решения, начиная с последнего элемента.

Обратный ход метода Жордана-Гаусса может быть реализован с помощью следующей функции.

```
using LinearAlgebra

function reverse_gauss(A::AbstractMatrix{T}, b::AbstractVector{T}) where T
    x = similar(b)
    N = size(A, 1)
    for k in 0:N-1
        x[N-k] = (b[N-k] - sum(A[N-k,N-k+1:end] .* x[N-k+1:end])) / A[N-k,N-k]
    end
    return x
end
```

Этот алгоритм можно улучшить (ускорить), если получать ссылки на срезы матрицы с помощью макроса `@view` или макроса `` `@views ``, а также с помощью мактоса `@inbounds` отменить контроль выхода индексов за пределы соответствующих массивов (но это, последнее, стоит делать только после того, как код функции будет полностью отлажен).

```
function reverse_gauss(A::AbstractMatrix{T}, b::AbstractVector{T}) where T
    x = similar(b)
    N = size(A, 1)
    @inbounds for k in 0:N-1
        @views x[N-k] = (b[N-k] - sum(A[N-k,N-k+1:end] .* x[N-k+1:end])) / A[N-k,N-
k]
    end
    return x
end
```

Здесь один макрос `@views` заменяет многократное использование макроса `@view`. С макросом `@view` строка в теле цикла выглядела бы так

```
 x[N-k] = (b[N-k] - sum(@view(A[N-k,N-k+1:end]) .* @view(x[N-k+1:end]))) / A[N-k,N-
k]
```

Кроме того `@inbounds` перед заголовком цикла можно перенести в любую отдельную стоку тела цикла.
В данном случае большой разницы нет в том, где поместить этот макрос, но если бы в теле цикла было несколько строк с индексными выражениями, то вариант с записью перед телом цикла выглядит более компактным.

Далее, этот код можно записать ещё с помощью вспомогательной функции `sumprod`:

```
function reverse_gauss(A::AbstractMatrix{T}, b::AbstractVector{T}) where T
    x = similar(b)
    N = size(A, 1)
    for k in 0:N-1
        x[N-k] = (b[N-k] - sumprod(@view(A[N-k,N-k+1:end]), @view(x[N-k+1:end]))) /
A[N-k,N-k]
    end
    return x
end


@inline function sumprod(A::AbstractVector{T},B::AbstractVector{T}) where T
    s = T(0)
    @inbounds for i in eachindex(A)
```

```
        s += A[i]*B[i]
    end
    return s
end
```

При этом, во-первых, здесь можно использовать макрос `@inline`, который исключит накладные расходы на вызовы вспомогательной функции `sumprod`, т.е. этот макрос указывает компилятору, что не нужно функцию `sumprod` компилировать отдельно, а нужно просто всюду заменять вызовы этой функции её телом, и после этого уже компилировать только вызывающую функцию.

Кроме того функцию `sumprod` можно оптимизировать, если две операции, выполняемые в цикле в теле этой функции на одну трехместную операцию, называемую `fma`:

```
@inline function sumprod(A::AbstractVector{T}, B::AbstractVector{T}) where T
    s = T(0)
    @inbounds for i in eachindex(A)
        s = fma(A[i], B[i], s)
    end
    return s
end
```

Использованная здесь функция `fma(x, y, z)` вычисляет выражение `x*y+z`:

```
help?> fma
  fma(x, y, z)

  Computes x*y+z without rounding the intermediate result x*y. On some systems this
is significantly more expensive than . fma is used to improve
  accuracy in certain algorithms. See muladd.
```

Дело в том, что среди команд современных процессоров есть специальная команда, аппаратно выполняющая сразу два арифметических действия x*y+z. Использование этой команды вместо двух отдельных операций повышает производительность и уменьшает погрешность соответствующего результата.

Замечание. Типы аргументов функций `AbstractMatrix`, `AbstractArray` обеспечивают большую универсальност по сравненю с тем, если бы типы аргументов функций были `Matrix` и `Vector`, соответсвенно.

Для тестирования алгоритма можно воспользоваться, например, следующим генератором случайных верхне-треугольных матриц

```
function random_upper_triangular(N::Integer)
    A = randn(N,N)
    _, A = lu(A)
    return A
end
```

Здесь использована функция `lu(A)` из пакета `LinearAlgebra`, которая раскладывает произвольную матриуцу A в произведение нижне-треуголной и верхне-треугольной (нам нужна только верхне-треугольная матрица из этого разложения).
Встроенная функция `randn` формирует случаюную матрицу требуемых размеров.

Приведение расширенной матрицы СЛАУ к ступенчатому виду

Приведение матрицы к ступенчатому виду, основанное на использовании элементарных преобразований строк, может быть реализована с помощью следующей функции.

```
function transform_to_steps!(A::AbstractMatrix; epsilon = 1e-7)
   @inbounds for k ∈ 1:size(A, 1)
        absval, Δk = findmax(abs, @view(A[k:end,k]))
        (absval <= epsilon) && throw("Вырожденая матрица")
        Δk > 1 && swap!(@view(A[k,k:end]), @view(A[k+Δk-1,k:end]))
        for i ∈ k+1:size(A,1)
            t = A[i,k]/A[k,k]
            @. @views A[i,k:end] = A[i,k:end] - t * A[k,k:end]

        end
    end
    return A
end
```

В этом коде важно отметить следующие моменты.

1. Через именованный параметр `epsilon` функция получает значение малой величины, которая используется для контроля не равенства нулю очередного диагонального элемента ступенчатой матрицы. Тут проблема в том, что в результате накопления ошибок получаемые малые значения на диагонали в действительности могут означать их равенство нулю. Параметр `epsilon` содержит "пороговое" значение для таких "нулей" (его значение по умолчанию принято равным `1e-7` в предположении, что порядок максимального значения элеменов матрицы равен нулю, и что вычисления производятся в формате `Float64`).

2. Макрос `@.` используется вместо того, чтобы в соответсвующей строчке каждую операцию записывать с точкой. Т.е. вместо соответсвующей строки можно было бы написать

```
A[i,k:end] .= A[i,k:end] .- t .* A[k,k:end]
```

(макрос `@view` опущен, чтобы не отвлекал внимания). В любом случае эти опрерации с точкой обеспечивают поэлементные вычисления "на месте", без создания буферных массивов (как было бы в данном случае без использования опереаций с точками).

3. Каждый раз при обнулении части столбца матрицы под её `k`-ым диагональным элементом, в соответствующей части `k`-го столбца ищется не просто ненулевой элемент, а элемент наибольший по модулю. После чего строка, содержащая этот наибольший по модулю элемент меняется местом с `k`-ой строкой. Это делается для того, чтобы коэффициет `t`, на который домножатся очередная строка при выполенеии соответсвующего элементарного преобразования строк, по модулю не превосходил бы единицу. В противном случае абсолютные погрешности элементов матрицы (неизбежно имеющие место вследствие особенностей арифметики с плавающей точкой) будут нарастать, что может привести к численной неустойчивости алгоритма. В самом деле, если обозначить через Δ - погрешность элемента матрицы $A[i,j]$, т.е. если (i,j)-ый элемент матрицы равен $A[i,j]+\Delta$, то в результате умножения этого элемента на коэффициент t величина погрешности результата умножения будет пропорциональна t. И если по абсолютной величине коэффициент t гарантированно не превосходит 1, то это будет сдерживать накопление ошибок. В противном случае, эффект накопления ошибок может привести к катастрофическим результатам, т.е. вычислительный процесс может оказаться численно не устойчивым.

Индекс (и абсолютное значение) очередного ведущего элемента (который после очередной перестановки строк должен оказаться на позиции `k`-го диагонального элемента), возвращается используемой здесь встроенной функцией `findmax`. Эта функция, кроме найденного максимального значения (в данном случае - абсолютного значения) элемента массива возвращает его индекс. Однако этот индекс отсчитывается от начала передаваемого в функцию срезе, а не от начала столбца матрицы, поэтому в дальнейшем это надо учитывать. Это учтено в строчке

```
Δk > 1 && swap!(@view(A[k,k:end]), @view(A[k+Δk-1,k:end]))
```

Эта строчка эквивалентна следующему коду

```
if Δk > 1
    swap!(@view(A[k,k:end]), @view(A[k+Δk-1,k:end]))
end
```

Здесь используется ещё одна вспомогательная функция

```
@inline function swap!(A,B)
    @inbounds for i in eachindex(A)
        A[i], B[i] = B[i], A[i]
    end
end
```

Решение СЛАУ

```
function solve_sla(A::AbstractMatrix{T}, b::AbstractVector{T}) where T
    Ab = [A b]
    transform_to_steps!(Ab; epsilon = 10sqrt(eps(T)) maximum(abs,A))
    return reverse_gauss(Ab)
end
```

Стоит обратить внимание, что величина "порга нуля" при приведении матрицы к ступенчатому виду принята равной `10sqrt(eps(T))*maximum(abs,A)`. Таким образом, если `T=Float64`, то порог нуля будет соответсвовать значениям на 7 порядков меньшим максимального значения матрицы.

Использованная здесь встроенная функция `eps` возвращает значение "машинного эпсилон" заданного типа с плавающей точкой.

Влияние на эффективность (производительность) кода порядок индексов многомерных массивов, в котором осуществляется их перебор во вложенных циклах

Допустим требуется просуммировать элементы каждой строки матрицы `A`. Это можно сделать следующими двумя способами

```
s=zeros(size(A,1))
for i in 1:size(A,1)
    for j in 1:size(A,2)
        s[i]+=A[i,j]
    end
end
```

```
s=zeros(size(A,1))
for j in 1:size(A,2)
    for i in 1:size(A,1)
        s[i]+=A[i,j]
    end
end
```

Спрашивается, какой из этих способов лучше (эффективнее)?

Ответ на этот вопрос зависит от того, в какой последовательности элементы массива (плотного) размещаются в памяти компьютера. В разных языках программирования этот порядок разный. Например, в языке C/C++, в памяти сначала следуют элементы первой строки, затем элементы второй строки, и т.д., в конце следуют элементы последнней строки. То же относится к массивам библиотеки `numpy` для языка `Python` (в самом языке `Python` плотные массивы вообще отсутсвуют).

В языке Julia (как и в языках FORTRAN, MATLAB), массивы в памяти размещаются наоборот, по столбцам. Т.е. сначала следуют элементы первого столбца, затем - второго, и т.д.

Если речь идет о многомерных массивах, с числом индексов большем двух, то та все аналогично. Например, в случае 3-х мерных массивов, сначала в памяти размещается первый слой 3-х мерного массива (в данном случае "слой" - это матрица), затем второй слой, и т.д. Элементы каждого слоя (матрицы) размещаются в последовательности, определяемой правилом, принятым в том или ином языке программирования.

Порядок следования элементов массивов в памяти оказывает влияние на производительность из-за того, что процессор компьютера имеет так называемую внутреннюю кеш-память (имеется даже целая иерархия кеш-памяти различного уровня). Эта кеш-память имеет намного более высокое быстродействие по отношению к обычной памяти. Поэтому процессор стремится размещать в кеш-памяти те данные, с которыми в данный момент производятся интенсивные вычисления. При этом стратегия, которой придерживается процессор, иногда приводит к ошибкам, называемым промахами кеша, заключающимися в том, что в кеш оказываются помещенными не те данные, которые на самом деле требуются в данный момент. Это приводит к замедлению вычислений.

При этом в кеш-память могут копироваться только непрерывные области памяти. Таким образом, если в памяти массив размещен по столбцам, а алгоритм осуществляет в данный момент последовательный перебор элементов некоторой

строки матрицы, то при большом размере матрицы промахи кеша будут гарантированы.

Исходя из сказанного, исходя из требования повышения производительности, при программировании на языке Julia перебор индексов строк матрицы должен всегда осуществляться во внутреннем цикле, а индексы столбцов - во внешнем. При программировании же на языках C/C++ - ровно наоборот.

Сказанное также означает, что приведенный выше код, реализующий метод Жордана-Гасса, реализован не самым лучшим образом. Эффективнее было бы представлять матрицу СЛАУ в транспонированном виде, по отношению к тому, как это было сделано.

Если теперь переписать всю программу соответсвующим образом, то, на самом деле, это не будет означать, что сначала надо сформировать матрицу, в строках которой содержатся коэффициенты отдельных уравнений СЛАУ, а затем эту матрицу транспонировать. Просто матрицу с коэффициентами уравнений надо будет сразу формировать требуемым образом, т.е. размещать коэффициенты отдельных уравнений в столбцах, а не в строках.