Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Глазачев Владимир Александрович

Использование методов интервального анализа в некоторых задачах линейной алгебры

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., академик Матиясевич Ю. В.

Рецензент:
д. ф.-м. н., профессор Нестеров В. М.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Vladimir Glazachev

Application of Interval Analysis techniques to some Linear Algebra problems

Bachelor's Thesis

Scientific supervisor:
academician Yuri Matiyasevich

Reviewer:
professor Vyacheslav Nesterov

Saint-Petersburg
2016

Оглавление

Введение

Величины с плавающей точкой являются одним из основных типов данных при математических вычислениях. Однако, они не всегда хорошо справляются с возложенной на них задачей быть аналогом вещественных чисел. Ошибка может быстро накапливаться и вычисленное значение может быть сколь угодно удаленным от правильного результата. Таким образом, необходима возможность автоматизации контроля за погрешностью при расчетах на компьютере.

Рассмотрим причины появления неточностей. Традиционно принято их делить на три класса:

1. погрешности в начальных данных;
2. погрешности метода;
3. погрешности округления.

Мы не можем автоматически контролировать погрешности метода, этим занимается отдельный раздел математики — вычислительные методы. В массовых языках обычно возможности указывать направление округления. Также они не позволяют учитывать погрешности в данных.

Интервальные методы применяются еще со времен Архимеда. Начало развития современного интервального анализа связывается с выходом книги "Interval Analysis" [7] Мура в 1966 году. Интервальное число представляет собой интервал, в котором гарантированно находится истинное значение. Далее определяются операции, результат которых также является гарантированным.

Таким образом интервалы позволяют одновременно представлять приближенное значение и его погрешность, а с проблемами округления можно бороться с помощью направленных округлений.

Раньше развитие интервального анализа затормаживалось из-за необходимости изучать спецификации конкретных машин, так как не существовало стандартов на вещественные числа. С появлением в 1985 году

IEEE 754 [3] стандарта и различных библиотек для работы с числами с произвольной точностью эта проблема сошла на нет.

Интервальные библиотеки в том или ином виде существуют почти для всех популярных языков программирования (INTLAB, Boost interval, libieeep1788, C-XSC, Pascal-XSC, Arb и многие другие). В таких системах компьютерной алгебры как Maple, Mathematica, MuPAD интервалы являются встроенными типами. Недавно вышел IEEE 1788 [4] стандарт на интервальные вычисления. В основном реализации интервальной арифметики предоставляют собой набор стандартных операций и функций без специальных методов улучшения оценки значения ошибки (таких как, например, обобщенный интервальный или апостериорно-интервальный анализ). Это связано с тем, что такие методы значительно увеличивают трудоемкость вычисления.

В данной работе рассматривается применимость апостериорно-интервального метода к таким задачам линейной алгебры как вычисление определителя и решение линейных систем. Также приводится описание реализации библиотеки для апостериорно-интервальных вычислений в произвольных программах.

1. Постановка задачи

Целью данной работы является исследование возможности применения методов интервального анализа в задачах линейной алгебры. В ходе выполнения работы необходимо решить следующие задачи:

1. Изучение теоретических основ интервального анализа, а также подходов к уточнению размера итогового интервала в ходе интервальных вычислений;
2. Реализация основных операций интервального анализа;
3. Реализация следующих задач линейной алгебры в терминах интервального анализа:
   - Вычисление определителя;
   - Решение системы линейных уравнений.
4. Реализация апостериорного подхода к уточнению ошибки для этих задач;
5. Сравнение традиционного и апостериорного подходов.

2. Обзор

Интервальная арифметика была разработана в качестве подхода к контролю ошибок округления для получения надежного результата. В то время как обычные вычисления производятся над отдельными числами, в интервальной арифметике все операции выполняются над интервалами. В этой главе будут описаны математические основы интервального анализа.

Для более детального знакомства с темой следует изучить следующие книги [11], [8].

Во время традиционных интервальный вычислений результирующий интервал часто оказывается слишком большим, что сильно снижает ценность такого ответа. Для уменьшения итоговой оценки были разработаны специальные методы уточнения ошибки, дающие более узкий результат ценой дополнительных вычислений. По большей части будет рассматриваться метод апостериорного уточнения ошибки описанный в работах [13], [6].

Предложенные методы также могут быть применимы в задачах, в которых мы имеем неопределенность уже на входе — например, измерения, полученные физическим прибором с известной погрешностью. Однако из-за специфики методов уточнения ошибок для их применимости накладываются дополнительные ограничения на размер входных интервалов.

2.1. Интервальный анализ

Будем обозначать замкнутые *интервалы* $\boldsymbol{x} = [a, b] = \{x \in \mathbb{R} \,|\, a \leq x \leq b\}$. Множество всех замкнутых интервалов $\mathbb{I} = \{[a, b] \;\; | \, a \leq b\}$. Таким образом, обычные числа $a \in \mathbb{R}$ будут представляться вырожденным интервалом $[a, a]$.

Главной идеей интервального анализа является продолжение обычных арифметических операций над вещественными числами на интервалы из $\mathbb{I}$. Для любых интервалов $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{I}$ и арифметических операций

$\circ \in \{+, -, \cdot, /\}$ имеем

$$\boldsymbol{x} \circ \boldsymbol{y} = x \circ y \,|\, x \in \boldsymbol{x}, y \in \boldsymbol{y} \tag{1}$$

Для удобства обозначим $\boldsymbol{x} = [\underline{x}, \overline{x}], \boldsymbol{y} = [\underline{y}, \overline{y}]$. Тогда можем переписать операции из (1) как:

$$\begin{aligned}
\boldsymbol{x} + \boldsymbol{y} &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
\boldsymbol{x} - \boldsymbol{y} &= [\underline{x} - \overline{y}, \underline{x} + \overline{y}] \\
\boldsymbol{x} \cdot \boldsymbol{y} &= [\min(\underline{x}\underline{y}, \overline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \overline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\overline{y})] \\
\boldsymbol{x} \,/\, \boldsymbol{y} &= [\underline{x}, \overline{x}] \cdot [\frac{1}{\overline{y}}, \frac{1}{\underline{y}}], \text{ если } 0 \notin \boldsymbol{y}
\end{aligned}$$

Рассмотрим *центр* интервала $m(\boldsymbol{x}) = (\underline{x} + \overline{x})/2$ и *ширину* $w(\boldsymbol{x}) = \overline{x} - \underline{x}$. Тогда интервал может быть переобозначен как (v, e), где $v = m(\boldsymbol{x})$, а $e = w(\boldsymbol{x})/2$. Операции определим следующим образом:

$$(v_1, e_1) + (v_2, e_2) = (v_1 + v_2, e_1 + e_2) \tag{2}$$

$$(v_1, e_1) - (v_2, e_2) = (v_1 - v_2, e_1 + e_2) \tag{3}$$

$$(v_1, e_1) \cdot (v_2, e_2) = (v_1 \cdot v_2, |v_1|e_2 + e_1|v_2| + e_1 e_2) \tag{4}$$

$$(v_1, e_1) \,/\, (v_2, e_2) = (\frac{v_1}{v_2}, \frac{e_1 + e_2 + \frac{v_1}{v_2}}{|v_2| - e_2}) \tag{5}$$

Таким образом, заменив обычные числа на интервалы одним из предложенных способов, мы получаем возможность работать с погрешностями во входных данных. Каждый из описанных способов представления интервалов имеют преимущества и недостатки. Например, в первом способе ничто не мешает нам добавить в качестве границы бесконечности и иметь интервалы вида $[a, +\infty)$. Также он дает более узкий интервал для умножения. Второй же способ удобнее с вычислительной точки зрения. В дальнейшем будет использоваться именно второй подход и обозначения интервалов вида $x = (v, e)$, представляющих собой естественную структуру для значения и его возможной ошибки. Обозначим $err(x) = e$ и $val(x) = v$. Такое представление может быть эффективнее и с вычислительной точки зрения — появляется

возможность применять различные типы для значения и ошибки. Так, например, устроена реализация интервальных значений в библиотеке Arb [5].

2.2. Оптимальная оценка ошибки

В результате интервальных вычислений часто получаются очень широкие интервалы. Отчасти это связано с ошибками во входных данных и округлениями, что позволяет нам иметь гарантированный результат. Но есть и погрешности другого рода, вызванные недостаточно полным использованием входной информации. Например, при вычислении $x - x$ должен получиться 0, но по формуле (3) имеем $x - x = (v, e) - (v, e) = (0, 2e) \neq 0$. Получается, что мы теряем информацию о том, что операнды арифметической операции взаимозависимы.

Рассмотрим теперь оценку ошибки при интервальных вычислениях. Пусть необходимо вычислить функцию $Y(x_1, x_2, ..., x_n)$, где $x_i = (v_i, e_i)$, задаваемую программой составленной из операций (2)-(5) и возвращающую интервал $y = (v, e)$. Вычисленный интервал y будет обладать следующим свойством:

$$|Y(r_1, r_2, ..., r_n) - v| \leq e,$$

для любых вещественных $r_1, r_2, ..., r_n$ таких что $|r_i - v_i| < e_i$.

Наименьшим допустимым значением e будет:

$$\Delta(\bar{v}, \bar{e}) = \Delta(v_1, ..., v_n, e_1, ..., e_n) = \max_{|r_i - v_i| < e_i} |Y(r_1, ..., r_n) - Y(v_1, ..., v_n)| \quad (6)$$

На практике отклонение вычисленного значения v от истинного обычно гораздо меньше $\Delta(\bar{v}, \bar{e})$, поскольку погрешности редко достигают своих граничных значений, а также могут компенсировать друг друга. Однако значение $\Delta(\bar{v}, \bar{e})$ дает нам строгую оценку. Значение же e полученное интервальным анализом может в любое число раз превосходить эту оценку.

Нахождение точного значения $\Delta(\bar{v}, \bar{e})$ являются NP-трудной задачей

для многих интервальных алгоритмов. В частности, таковыми являются рассматриваемые в работе задачи вычисления определителя и решения линейных систем [12], [9]. Поэтому построение методов, дающих лучшую оценку, чем традиционный интервальный анализ, не сильно проигрывая при этом в производительности, является важной задачей.

2.3. Апостериорный интервальный анализ

Из того, что $Y(\bar{r}) \subseteq Y(\bar{r})+(\bar{v}-\bar{r})Y'(\bar{v})$ имеем, что для малых значений $e_1, ..., e_n$ верно

$$Y(\bar{r}) - Y(\bar{v}) \approx \sum_{i=1}^{n} \frac{\partial Y(\bar{v})}{\partial v_i}(r_i - v_i). \tag{7}$$

Получаем оценку ошибки

$$\Delta(\bar{v}, \bar{e}) \approx \sum_{i=1}^{n} \left| \frac{\partial Y(\bar{v})}{\partial v_i} \right| e_i. \tag{8}$$

Будем называть решение *ассимптотически оптимальным*, если

$$\frac{e}{\Delta(\bar{v}, \bar{e})} = \frac{\sum_{i=1}^{n} \left| \frac{\partial Y(\bar{v})}{\partial v_i} \right| e_i}{\Delta(\bar{v}, \bar{e})} \xrightarrow[\bar{e}\to\bar{0}]{} 1 \tag{9}$$

для всех $v_1, ..., v_n$ таких, что $Y(\bar{v})$ определено и $\frac{\partial Y(\bar{v})}{\partial v_i} \neq 0$.

Можно показать, что традиционный интервальный анализ не дает асимптотически оптимального решения, однако он удовлетворяет более слабому условию

$$\frac{e}{\Delta(\bar{v}, \bar{e})} = O(1). \tag{10}$$

То есть решение в константное число раз хуже оптимального. А сокращение входных погрешностей в m раз ведет к сокращению в m раз и выходных погрешностей.

Асимптотически оптимальное решение было предложено Хансеном в [2] где он ввел *обобщенную интервальную арифметику*. Также был предложен метод вычисления частных производных в (7). В данной работе не будет приводиться подробное сравнение предложенной реали-

зации с обобщенной интервальной арифметикой, поскольку последняя имеет бóльшую временную сложность по причине вычислений, которых можно избежать. В ходе вычислений находятся не только частные производные конечного результата, но и все частные производные всех промежуточных результатов (если начальные метод имеет сложность $T(x_1, ..., x_n)$, тогда традиционная интервальная арифметика будет иметь сложность $O(T(x_1, ..., x_n))$, а обобщенная $O(nT(x_1, ..., x_n))$. Например, при вычислении определителя матрицы $n \times n$ сложность вычисления в обобщенной интервальной арифметике будет $O(n^5)$.

Однако, в [1] было показано, что сложность вычисления функции и всех ее частных производных не превосходит сложности вычисления самой функции более чем в константу раз. Рассмотрим способ вычисления градиента, обладающего таким свойством для программы, вычисляющей рациональную функцию f от n переменных $x_1, ..., x_n$. Функция может быть вычислена как последовательность вида:

$$\begin{gathered}(x_1 = x_1^0, ..., x_n = x_n^0);\\ x_{n+1} := x_{i_{n+1}} \circ_{n+1} x_{j_{n+q}};\\ ...\\ x_l := x_{i_l} \circ_l x_{j_l};\\ ...\\ x_m := x_{i_m} \circ_m x_{j_m},\end{gathered} \tag{11}$$

где x_m — значение функции, $n \leq l \leq m$, $l > l_i$, $l > l_j$, $\circ_l$ — арифметическая операция, а x_k — начальные или промежуточные значения.

Допустим, что $x_1, ..., x_n$ — функции от переменной t, такие, что в нуле они имеют значения $x_1^0, ..., x_n^0$ соответственно. Рассмотрим уравнение вида

$$\frac{\partial f}{\partial t} = \frac{\partial z_1}{\partial x_1} + ... + \frac{\partial z_l}{\partial x_l} = \sum_{i=1}^{l} z_i \frac{\partial x_i}{\partial t}. \tag{12}$$

При $l = n$ имеется единственное решение $z_i = \frac{\partial f}{\partial x_i}$. При $l = m$ существует

другое решение $z_m = 1$, $z_i = 0$, $i < m$. Добавим строчки

$$\begin{aligned} z_1 &:= 0; \\ &... \\ z_{m-1} &:= 0; \\ z_m &:= 1, \end{aligned} \tag{13}$$

к программе (11). Теперь, последовательно будем добавлять строчки к полученной программе для $l = m...n$ в зависимости от $\circ_l$. А именно:

если $\circ_l = +$, то добавляются строки

$$\begin{aligned} z_i &:= z_i + z_l; \\ z_j &:= z_j + z_l, \end{aligned} \tag{14}$$

если $\circ_l = -$, то добавляются строки

$$\begin{aligned} z_i &:= z_i + z_l; \\ z_j &:= z_j - z_l, \end{aligned} \tag{15}$$

если $\circ_l = *$, то добавляются строки

$$\begin{aligned} z_i &:= z_i + z_l * x_j; \\ z_j &:= z_j + z_l * x_i, \end{aligned} \tag{16}$$

если $\circ_l = /$, то добавляются строки

$$\begin{aligned} z_i &:= z_i + z_l / x_j; \\ z_j &:= z_j - z_l * x_l / x_j^2, \end{aligned} \tag{17}$$

Получив финальную программу при $l = n$ получим решение — значение всех частных производных. Новой оценкой ошибки $err(x_m)$ будет:

$$err(x_m) = \sum_{i=1}^{n} |z_i| \, err(x_i). \tag{18}$$

При таком вычислении может показаться, что полученные значения z_i, посчитанные традиционными интервальными операциями, бу-

дут иметь большу́ю ошибку и вместо уточнения одной выходной величины нам необходимо уточнять и z_i. Однако, в силу (10) эти значения будут иметь лишь константное ухудшение по сравнению с асимптотически оптимальным и при $\bar{e} \to 0$ ошибка $err(z_i) \to 0$, так что полученная предложенным методом оценка ошибки будет асимптотически оптимальной.

Приведенные метод требует не более 5 дополнительных операций на каждую операцию в начальной программе. Таким образом, имеем лишь линейное замедление по сравнению с традиционным интервальным анализом.

Данный метод был предложен Ю.В.Матиясевичем в [13]. Этот метод был назван апостериорным интервальным анализом, и это означает то, что, в от отличии интервальной арифметики, этот метод не позволяет закончить вычисление ошибки в момент окончания вычисления функции, поскольку ему требуются значения всех промежуточных вычислений в порядке обратном их вычислению.

Таким образом, апостериорно-интервальное вычисление состоит из двух фаз. Первая — традиционное интервальное вычисление с сохранением всех промежуточных величин. Вторая фаза — вычисление значения производных и уточненной оценки погрешности.

3. Реализация

Реализация представляет собой библиотеку на C++ и имеет следующую структуру:

- Класс для работы с традиционными интервалами;
- Класс для объекта управления динамическим апостериорным вычислением;
- Статические методы для вычисления определителя и решения линейных систем;
- Примеры программ.

В этой части будут описаны первые два пункта. Статические методы будут описаны в следующей главе.

3.1. Традиционный интервальный анализ

Поскольку эффективная реализация интервальных вычислений является отдельной трудоемкой задачей, интервальная арифметика реализована как оболочка над интервальными типами библиотеки Arb [5]. Arb является довольно низкоуровневой библиотекой и программы, написанные с использованием его интерфейсов, получаются громоздкими. Используя перегрузку операторов и автоматически вызываемые в конструкторе/деструкторе методы выделения/возвращения памяти можно значительно сократить код программы и увеличить читаемость.

В программе, написанной с перегрузкой операторов, возникает один недостаток — в Arb для каждой арифметической операции существует возможность указывать с какой точностью (количество бит) она будет вычислена. В предложенной реализации значение точности может быть задано пользователем, но оно является общим для всех вычислений.

На данный момент реализованы только основные арифметические операции, методы для сравнения и вывода.

3.2. Динамический апостериорно-интервальный анализ

В работах [14], [13] описывается реализация апостериорно-интервального анализа, модифицирующего текст программы для построения второго шага апостериорных вычислений. Также в работе [13] предлагается возможная архитектура вычислительной машины, названной апостериорно-интарвальной машиной, автоматически производящей второй этап. Динамическая реализация представляет собой класс структур, позволяющих моделировать эту вычислительную машину на обычном компьютере.

Основными единицами являются традиционные интервалы из предыдущего пункта. Арифметические операции выполняются в соответствии с (2)—(5). Проблемы округления переложены на внутреннее устройство библиотеки Arb, так что мы их затрагивать не будем.

Имеется две группы команд — первой и второй вычислительной фазы. Команды первой фазы — это арифметические операции $+, -, *, /$. Вместо непосредственно идентификаторов переменных они принимают адреса в памяти `mem`. Будем обозначать взятие значения элемента по адресу `addr` в `mem` как $\langle$`addr`$\rangle$.

Для хранения программы второй фазы используется специальная стековая память (в нашем случае она моделируется в обычной оперативной памяти). Также имеется один сумматор `s`. Во время второй фазы используются три команды — `NULL`, `INULL` и `CORR`. Их действия заключаются в следующем:

`NULL addr` — одноадресная команда, записывает в сумматор `s` значение $\langle$`addr`$\rangle$ и обнуляет память по адресу `addr`.

`INULL addr` — одноадресная команда, записывает в сумматор `s` значение по модулю $|\langle$`addr`$\rangle|$ и обнуляет память по адресу `addr`.

`CORR addr interval` — двухадресная команда, записывает в ячейку по адресу `addr` значение $\langle$`add`$\rangle$+`s`$*$`interval`.

Рассмотрим теперь арифметические команды первой фазы. Будем обозначать их как `ADD`, `SUB`, `MUL`, `DIV`. Эти команды являются трехад-

ресными и имеют следующий интерфейс:

$$\texttt{COMMAND addr1 addr2 addr3}$$

будет производить следующую операцию

$$\texttt{addr1} \leftarrow \langle\texttt{addr2}\rangle \circ \langle\texttt{addr3}\rangle.$$

Помимо засылки значения в addr1, будут записываться команды в стек в зависимости от типа операции. А именно:

при исполнении команды ADD будут записаны

$$\begin{array}{ll} \texttt{CORR addr3} & \texttt{(1,0)} \\ \texttt{CORR addr2} & \texttt{(1,0)} \\ \texttt{NULL addr1} & \end{array} \tag{19}$$

при исполнении команды SUB будут записаны

$$\begin{array}{ll} \texttt{CORR addr3} & \texttt{(-1,0)} \\ \texttt{CORR addr2} & \texttt{(1,0)} \\ \texttt{NULL addr1} & \end{array} \tag{20}$$

при исполнении команды MUL будут записаны

$$\begin{array}{ll} \texttt{CORR addr3} & \langle\texttt{addr2}\rangle \\ \texttt{CORR addr2} & \langle\texttt{addr3}\rangle \\ \texttt{NULL addr1} & \end{array} \tag{21}$$

при исполнении команды DIV будут записаны

$$\begin{array}{ll} \texttt{CORR addr3} & \dfrac{-\langle\texttt{addr2}\rangle}{\langle\texttt{addr3}\rangle\langle\texttt{addr3}\rangle} \\ \texttt{CORR addr2} & \dfrac{\texttt{(1, 0)}}{\langle\texttt{addr3}\rangle} \\ \texttt{NULL addr1} & \end{array} \tag{22}$$

Перед началом исполнения программы первой фазы входные интервальные величины $x_1, ..., x_n$, $x_i = (v_i, e_i)$ заносятся в ячейки памяти

с номерами $1, ..., n$, а в стек записываются команды

```
CORR  1  (1, 0)
INULL 2
CORR  2  (1, 0)
INULL 3
  ...
CORR  n-1  (1, 0)
INULL n
CORR  n  (e_n, 0)
INULL n
  ...
CORR  1  (e_1, 0)
INULL 1
```
(23)

После исполнения первой фазы имеем посчитанное традиционным методом выходное значение y. Обычная память обнуляется и в ячейку, откуда было считано y, записывается интервал $(1, 0)$. После этого выполняются команды, записанные в стек в порядке, обратном поступлению. По окончанию этой фазы уточненной оценкой погрешности будет величина

$$val(\langle 1 \rangle) + err(\langle 1 \rangle).$$

При моделировании апостериорно-интервальной машины используется специальный объект, являющийся своего рода виртуальной машиной. Он содержит внутри себя выделенные в оперативной памяти вектора интервальных значений и команд для моделирования обычной и стековой памяти. А также предоставляет интерфейс для выполнения арифметических операций.

Для удобной работы с этим объектом была реализована дополнительная обертка над традиционными интервалами из предыдущего пункта. Этот прокси объект содержит внутри себя традиционный интервал и приписанный ему адрес в управляющем устройстве. Также реали-

зована перегрузка арифметических операторов, что позволяет пользователю применять эти объекты в своих вычислениях. Например, если был реализован шаблонный метод вычисления функции, внутри которого используются только основные арифметические операции и имеется одно выходное значение, то в результате вычислений с интервальным прокси-объектом автоматически будет произведено апостериорное уточнение ошибки.

Для функций с несколькими выходными переменными реализован дополнительный объект с перегрузкой операции присваивания. При присваивании он копирует управляющее устройство и производит уточнение ошибки для последней произведенной операции. Таким образом могут быть легко реализованы программы с несколькими выходными значениями.

4. Статические методы

Статические методы позволяют решить проблему использования дополнительной памяти, которая возникает в динамическом методе. Это происходит за счет восстановления значений на ходу. Возникает, однако, новая проблема — ожидаемая оценка ошибки будет больше из-за свойств операций над интервальными числами. Зато для методов с одним выходным значением мы получаем улучшение оценки почти бесплатно — метод имеет туже асимптотическую временную и пространственную сложность, что и в традиционной интервальной арифметике.

4.1. Общая схема статических методов

Для линейных программ, то есть программ не имеющих циклов, условных переходов и повторных присваиваний вида (11) статический метод будет заключаться в добавлении к коду программы строчек вида (13)—(18). Если необходимо получить текст программы апостериорного метода для произвольных программ так не получится. Конечно, можно пытаться сводить программы с повторными присваиванием к программам без повторного присваивания переименованием переменных и разворачивать циклы, если они имеют фиксированную длину. Исходный код таких программ будет быстро разрастаться и пространственная сложность станет не меньше временной сложности исходного алгоритма. Также такой подход не применим для большинства программ из-за параметризации длины входа.

Приведем метод описания второго шага апостериорного подхода позволяющий получать компактную и эффективную по памяти реализацию. Подробное описание предложенного метода автоматического дифференцирования стоит смотреть в [10].

Пусть программа содержит цикл:

```
for i := L to U do S,
```

тогда его инверсией будет

```
for i := U downto L do S^-1
```

.

Пусть теперь хотим обратить выражение, в котором есть общие операнды в левой и правой части, то есть происходит перезапись переменной. Выражение имеет вид:

$$x_t^{new} := x_t^{old} \circ x_r$$

или

$$x_t^{new} := x_l \circ x_t^{old},$$

тогда, в зависимости от $\circ$ выражением в инвертированной программе будет:

если $x_t^{new} := x_t^{old} + x_r$:

$$x_t^{old} := x_t^{new} - x_r; \quad dx_r := dx_r + dx_t; \quad dx_t^{new} := dx_t^{old};$$

если $x_t^{new} := x_t^{old} - x_r$:

$$x_t^{old} := x_t^{new} + x_r; \quad dx_r := dx_r - dx_t; \quad dx_t^{new} := dx_t^{old};$$

если $x_t^{new} := x_t^{old} * x_r$ и $x_r \neq 0$:

$$x_t^{old} := x_t^{new}/x_r; \quad dx_r := dx_r + dx_t * x_t^{old}; \quad dx_t^{new} := dx_t^{old} * x_r;$$

если $x_t^{new} := x_t^{old}/x_r$:

$$x_t^{old} := x_t^{new} * x_r; \quad dx_r := dx_r - dx_t * x_t^{new}/x_r; \quad dx_t^{new} := dx_t^{old}/x_r;$$

если $x_t^{new} := x_l + x_t^{old}$:

$$x_t^{old} := x_t^{new} - x_l; \quad dx_l := dx_l + dx_t; \quad dx_t^{new} := dx_t^{old};$$

если $x_t^{new} := x_l - x_t^{old}$:

$$x_t^{old} := x_l - x_t^{old}; \quad dx_l := dx_l + dx_t; \quad dx_t^{new} := -dx_t^{old};$$

если $x_t^{new} := x_l * x_t^{old}$ и $x_l \neq 0$:

$$x_t^{old} := x_t^{new}/x_l; \quad dx_l := dx_l + dx_t * x_t^{old}; \quad dx_t^{new} := dx_t^{old} * x_l;$$

если $x_t^{new} := x_l / x_t^{old}$:

$$x_t^{old} := x_l / x_t^{old}; \quad dx_l := dx_l + dx_t / x_t^{old}; \quad dx_t^{new} := -dx_t^{old} * x_t^{new} / x_t^{old};$$

Рассмотрим пример вычисления функции $f(x) = x^n$, которая может быть вычислена следующим образом:

```
p := 1;
for i := 1 to n do
  p := p * x;
```

По описанным выше правилам построим программу, вычисляющую $f'(x) = n * x^{n-1}$:

```
p := 1;
for i := 1 to n do
  p := p * x;
dx := 0;
dp := 1;
for i := n downto 1 do
  p := p / x;
  dx := dx + dp * p;
  dp := dp * x;
```

В результате в переменной `dp` будет находиться значение производной $f'(x)$ и при интервальных вычислениях для уточнения ошибки, полученного в ходе вычислений интервала x^n, необходимо значение `dp` умножить на ошибку входного интервала x.

В работе [10] приводится аналогичный пример для вычисления градиента определителя матрицы. Выделим отдельно этап обращения метода Гаусса (`GaussElimination`) в алгоритм 1 — `InverseGauss`. Имея матрицу частных производных можем найти итоговую оценку ошибки по алгоритму 2 — `EvalError`. Наконец, можем описать общую схему алгоритмов, использующих метод Гаусса — `CommonScheme` (Алго-

ритм 3). На первом шаге происходит вызов `GaussElimination`, приводящий матрицу к верхнетреугольному виду. Далее выполняются необходимые вычисления, например, нахождение определителя или решение линейной системы. Следующим этапом является инвертирование шага вычисления результирующего значения и, наконец, происходит вызов `InverseGauss` вычисляющий итоговый градиент. Последним шагом вычисляем новую оценку ошибки в `EvalError`.

Таким образом, если первым этапом алгоритма является применение метода Гаусса к матрице, то для статической реализации всего алгоритма необходимо реализовать метод (`InverseCompute`), инвертирующий основные вычисления (`Compute`) производимые для получения ответа после вызова метода Гаусса.

Алгоритм 1 `InverseGauss` — Обратный метод Гаусса

Вход: $A, dA \in \mathbb{I}^{n \times m}$; A — матрица входных данных после метода Гаусса; dA — подготовленная матрица производных
Выход: dA — матрица частных производных

1: **for** $i := n - 1$ **downto** 1 **do**
2: **for** $j := n$ **downto** $i + 1$ **do**
3: $dA_{i,j} := dA_{i,j} - dA_{j,i+1..m} \cdot A_{i,i+1..m}$;
4: $dA_{i,i+1..m} := dA_{i,i+1..m} - dA_{j,i+1..m} A_{j,i}$;
5:
6: $A_{j,i+1..m} := A_{j,i+1..m} + A_{i,j} * A_{i,i+1..m}$;
7:
8: $dA_{i,i} := dA_{i,i} - dA_{j,i} * A_{i,j} / A_{i,i}$;
9: $dA_{j,i} := dA_{j,i} / A_{i,i}$;
10:
11: $A_{j,i} := A_{j,i} * A_{i,i}$;
12: **end for**
13: **end for**
14: **return** dA;

4.2. Compute и InverseCompute для определителя

Рассмотрим теперь методы `Compute` (Алгоритм 4) и `InverseCompute` (Алгоритм 5) для вычисления определителя.

Алгоритм 2 `EvalError` — Вычисление итоговой ошибки

Вход: $A, dA \in \mathbb{I}^{n \times m}$; A — матрица входных данных; dA — матрица частных производных

Выход: e — значение ошибки

1: $e = 0$
2: **for** $i := 1$ **to** n **do**
3: **for** $j := 1$ **to** m **do**
4: $e = e + (|val(dA_{i,j})| + err(dA_{i,j})) * err(A_{i,j})$;
5: **end for**
6: **end for**
7: **return** e;

Алгоритм 3 `CommonScheme` — Общая схема вычисления итоговой ошибки

Вход: $A \in \mathbb{I}^{n \times m}$ — матрица входных данных

Выход: $x \in \mathbb{I}^{l}$ — выходные величины

1: $A_g :=$ GaussElimination(A);
2: $x :=$ Compute(A_g);
3: **for** $i := 1$ **to** l **do**
4: $dA :=$ InverseCompute(A_g, x, i);
5: InverseGauss(A_g, dA);
6: $err(x_i) =$ EvalError(A, dA);
7: **end for**
8: **return** x;

Алгоритм 4 `Compute` — Вычисление определителя

Вход: $A \in \mathbb{I}^{n \times m}$ — входная матрица после применение метода Гаусса

Выход: x — значение определителя

1: $x := 1$;
2: **for** $i := 1$ **to** n **do**
3: $x := x * A_{i,i}$;
4: **end for**
5: **return** x;

Алгоритм 5 `InverseCompute` — Схема подготовки матрицы dA для определителя

Вход: $A \in \mathbb{I}^{n \times m}$ — входная матрица после применение метода Гаусса; x — значение определителя

Выход: $dA \in \mathbb{I}^{n \times m}$ — подготовленная матрица частных производных

1: $dx := 1$;
2: $dA := 0$;
3: **for** $i := n$ **downto** 1 **do**
4: $\quad x := x / A_{i,i}$;
5: $\quad dA_{i,i} := dA_{i,i} + dx * x$;
6: $\quad dx = dx * A_{i,i}$;
7: **end for**
8: **return** dA;

Этот метод работает для метода Гаусса без выбора ведущего элемента. Для того, чтобы он подходил и для случая с выбором ведущего, необходимо в конце метода `Compute` домножить x на знак перестановки. В начале метода `InverseCompute` в момент инициализации dx его необходимо домножить на знак перестановки.

Получаем статические методы уточнения ошибки для вычисления определителя методом Гаусса с выбором ведущего и без. Оба метода требуют $O(n^3)$ арифметических операций и $O(n^2)$ памяти, что соответствует традиционному методу.

4.3. Compute и InverseCompute для решения линейных систем

Приведем теперь методы `Compute` (Алгоритм 6) и `InverseCompute` (Алгоритм 7) для решения линейных систем методом Гаусса.

Этот метод требует $O(n^2)$ арифметических операций и $O(n^2)$ дополнительной памяти. Итоговая сложность алгоритма `CommonScheme` будет $O(n^4)$ количества операций и $O(n^2)$ дополнительной памяти.

Алгоритм 6 `Compute` — Вычисление x

Вход: $A \in \mathbb{I}^{n \times n+1}$ — входная матрица после применение метода Гаусса;
Выход: $s \in \mathbb{I}^n$ — вектор решений

1: $x := 0$;
2: **for** $i := n$ **downto** 1 **do**
3: $\quad x_i := A_{i,n+1}$;
4: $\quad$ **for** $j := i + 1$ **to** n **do**
5: $\quad\quad x_i := x_i - A_{i,j} * x_j$;
6: $\quad$ **end for**
7: $\quad x_i = x_i / A_{i,i}$;
8: **end for**
9: **return** x;

Алгоритм 7 `InverseCompute` — Вычисление уточненного значения x_c

Вход: $A \in \mathbb{I}^{n \times n+1}$ — входная матрица после применение метода Гаусса;
$x \in \mathbb{I}^n$ — вектор решений; c — номер уточняемого решения
Выход: dA — Подготовленная матрица производных для x_i

1: $dxs := 0$;
2: $dxs_c := 1$;
3: $dA := 0$;
4: **for** $i := c$ **to** n **do**
5: $\quad t := x_i / A_{i,i}$;
6: $\quad dt := dxs_i / A_{i,i}$;
7: $\quad dA_{i,i} := dA_{i,i} - xs_i * A_{i,i}$;
8: $\quad dA_{i,n+1} := dt$;
9: $\quad$ **for** $j := n$ **downto** $i + 1$ **do**
10: $\quad\quad t := t + A_{i,j} * x_j$;
11: $\quad\quad dA_{i,j} = dA_{i,j} - xs_j * dt$;
12: $\quad\quad dxs_j = dxs_j - A_{i,j} * dt$;
13: $\quad$ **end for**
14: **end for**
15: **return** dA;

5. Сравнения подходов

В этой части приводятся таблицы сравнения теоретической сложности алгоритмов. Также приводятся экспериментальные результаты на случайно сгенерированных матрицах.

Замеры времени работы алгоритмов производились при помощи встроенной в C++ библиотеки chrono на машине со следующими техническими характеристиками: процессор Intel Core i7-4790 с частотой 3.6GHz, 8 Гб ОЗУ. Вычисления проводились с точностью в 1024 бит. Случайные числа генерировались равномерно из распределения на $[-5, 5]$.

5.1. Генерирование случайных матриц

Для проведения экспериментов необходимо иметь большое количество матриц для тестирования. При заполнении элементов матрицы случайными значениями определитель может быть очень большим или матрица может быть необратимой. Для того, чтобы генерировать случайные матрицы с фиксированным определителем воспользуемся функцией e^X — матричная экспонента квадратной матрицы X определяется как:

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k \tag{24}$$

Эта функция обладает следующим свойством — $det(e^X) = e^{tr(X)}$. Генерируя случайные матрицы A с фиксированным tr (например, нули на диагонали) и вычисляя e^A получаем различные матрицы с определителем, равным единице.

5.2. Определители

В таблице 1 приведена вычислительная сложность различных методов нахождения определителя. Методы с выбором и без выбора ведущего имеют одинаковую сложность, так что дополнительно не приводятся. Статический и динамический методы обладают одинаковой временной

Таблица 1: Вычислительная сложность для нахождения определителя

Метод	Временная сложность	Пространственная сложность
Традиционный	$O(n^3)$	$O(n^2)$
Динамический	$O(n^3)$	$O(n^3)$
Статический	$O(n^3)$	$O(n^2)$

Таблица 2: Ошибки для определителя различными методами

Метод	Значение ошибки
Оптимальный	2.1
Традиционный	10.9
Тр. с выбором ведущего	3.28
Динамический	2.56
Дин. с выбором ведущего	2.16
Статический	4.49
Стат. с выбором ведущего	2.31

сложностью, но статический метод использует на порядок меньше памяти. В практической реализации ожидается также, что статический метод будет работать быстрее динамического, из-за дополнительных затрат на работу управляющего устройства.

С другой стороны, динамический метод должен давать самый точный результат, далее идет статический — это происходит вследствие дополнительной накопленной ошибки во время восстановления промежуточных значений.

Рассмотрим пример вычисления определителя:

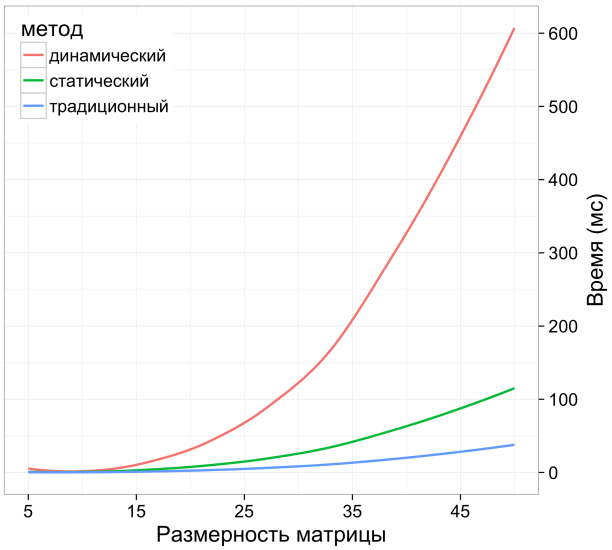
$$\begin{vmatrix} 4 & 7 & 8 \\ 6 & 4 & 6 \\ 7 & 3 & 10 \end{vmatrix} = -118$$

Если добавить к каждому элементу матрицы ошибку 0.01, то посчитав различными методами получим значения итоговых ошибок, указанные в таблице 2.

На графике 1 изображена зависимость времени работы различных методов от размера матрицы. Статический метод работает приблизи-

тельно в 3 раза медленнее традиционного, а динамический в 20 раз медленнее. Такой разрыв происходит по причине наивной реализации управляющего устройства. В дальнейшем динамический метод может быть значительно ускорен при более низкоуровневой реализации.

Рис. 1: Время вычисления определителя



Напомним, что из 10 следует, что традиционные вычисления имеют константное ухудшение по сравнению с асимптотически-оптимальными методами. На графике 2 изображено отношение значения ошибки, полученной традиционным методом, к значению ошибки, полученной апостериорным методом, то есть во сколько раз апостериорная ошибка меньше традиционной. Видно, что с увеличением размерности апостериорный метод дает намного более точный, по сравнению с традиционным, результат.

5.3. Линейные системы

В таблице 3 приведена вычислительная сложность различных методов решения линейных систем. Статический и динамический методы имеют одинаковую временную сложность, но статический метод имеет на порядок меньшую оценку используемой памяти.

Аналогично, динамический метод должен давать самый точный результат, далее идет статический — это происходит из-за дополнитель-

Рис. 2: Медианное значение отношения ошибки традиционных вычислений к апостериорным для определителя

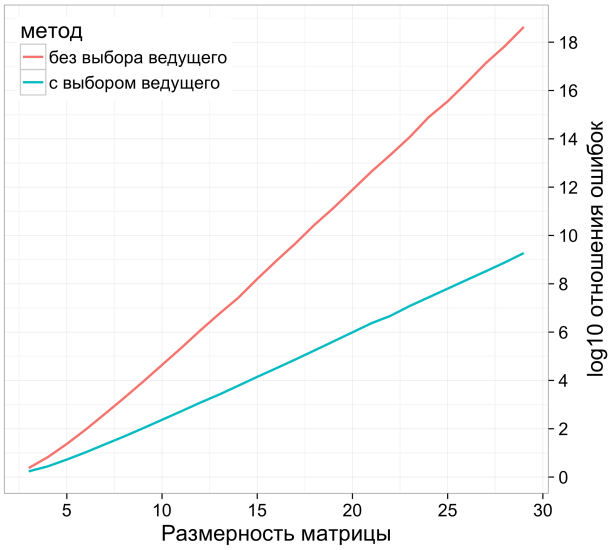


Таблица 3: Вычислительная сложность для решения линейных систем

Метод	Временная сложность	Пространственная сложность
Традиционный	$O(n^3)$	$O(n^2)$
Динамический	$O(n^4)$	$O(n^3)$
Статический	$O(n^4)$	$O(n^2)$

ной накопленной ошибки во время восстановления промежуточных значений.

На графике 3 изображено время работы различных методов в зависимости от размерности матрицы. Мы видим уже иную по сравнению с определителями картину, ведь методы уточнения ошибки требуют на порядок больше времени. Так, для системы размера 20×20 традиционный метод почти в 50 раз быстрее статического и в 180 раз быстрее динамического. На графике 4 приведено отношение значения ошибки, полученной традиционным методом, к значению ошибки, полученной апостериорным методом. Хорошо видно, что апостериорный метод дает на порядки меньшую оценку ошибки, чем традиционный.

Рис. 3: Время решения линейной системы

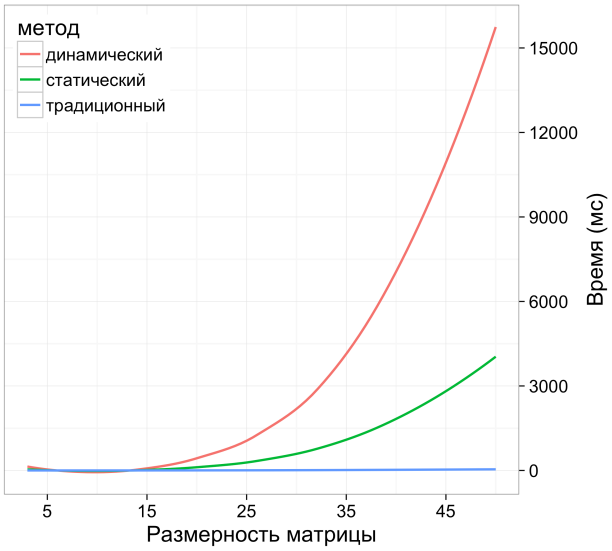
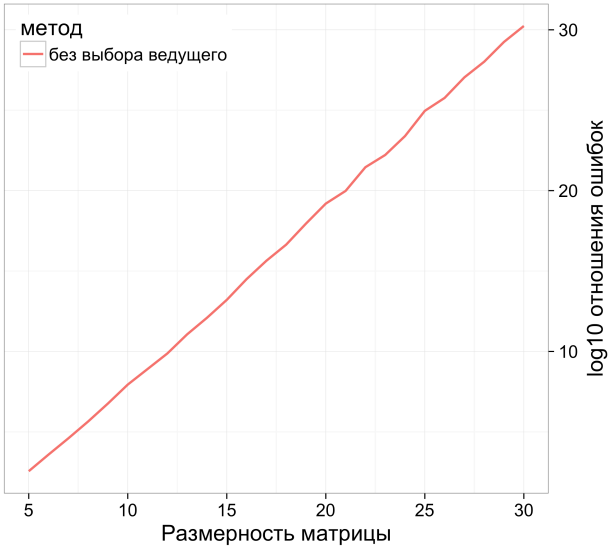


Рис. 4: Медианное значение отношения ошибки традиционных вычислений к апостериорным для линейных систем



Заключение

В результате данной работы были решены следующие задачи:

- исследованы подходы к работе с интервальными величинами, рассмотрены способы уточнения оценки итоговой ошибки;
- реализованы основные операции интервального анализа поверх библиотеки Arb;
- реализованы динамический и статический подход к апостериорному уточнению ошибки в задачах нахождения определителя и решения линейных систем;
- динамический подход реализован в качестве библиотеки и может быть легко использован в других приложениях;
- проведено теоретическое и экспериментальное сравнение подходов, показавшее, что, несмотря на бóльшие затраты по сравнению с традиционным интервальным анализом, апостериорный метод может быть эффективно использован в задачах линейной алгебры.

Дальнейшее развитие может происходить в нескольких направлениях.

Во первых, необходим более тщательный анализ и оптимальная реализация динамического метода. Это может на порядки сократить накладные расходы на использование управляющего устройства.

Во вторых, можно построить больше статических методов. Также полезна была бы возможность эффективно комбинировать динамический и статический методы.

Наконец, важной задаче является исследование возможности построения интервального языка с возможностью автоматической генерации статического этапа.

Список литературы

[1] Baur W., Strassen V. The complexity of partial derivatives // Theoretical computer science. — 1983. — Vol. 22, no. 3. — P. 317–330.

[2] Hansen E. A generalized interval arithmetic // Lecture Notes in Computer Science. — 1975. — Vol. 29. — P. 7–18.

[3] IEEE Standard for Floating-Point Arithmetic // IEEE Std. 754-1985. — 1985.

[4] IEEE Standard for Interval Arithmetic // IEEE Std. 1788-2015. — 2015.

[5] Johansson F. Arb: a C library for ball arithmetic // ACM Communications in Computer Algebra. — 2013. — Vol. 47, no. 4. — P. 166–169.

[6] Matiyasevich Yu. A posteriori interval analysis // Lecture Notes in Computer Science. — 1985. — Vol. 204. — P. 328–334.

[7] Moore R. Interval Analysis. — Prentice-Hall, 1996.

[8] Moore R., Kearfott R., Cloud M. Introduction to Interval Analysis. — Society for Industrial and Applied Mathematics Philadelphia, 2009.

[9] P. Kreinovich V., Lakeyev A., Rohn J., Kahl. Computational Complexity and Feasibility of Data Processing and Interval Computations. — Springer-Science Busines+Media, 1998.

[10] Shiriaev D. Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment : Ph. D. thesis / D. Shiriaev ; Karlsruhe University. — 1993.

[11] Алефельд Г., Херцбергер Ю. Введение в интервальные вычисления: Пер. с англ. — Мир, 1987.

[12] Гаганов А.А. О сложности вычисления интервала значений полинома от многих переменных // Кибернетика. — 1985. — Vol. 4. — P. 6–8.

[13] Матиясевич Ю.В. Вещественные числа и ЭВМ // Кибернетика и вычислительная техника. — 1986. — Vol. 2. — P. 104–133.

[14] Мусаев Э.А. Расширение апостериорно-интервального анализа на случай произвольных программ и его опытная реализация : Дисс... кандидата наук / Э.А. Мусаев ; Ленинградский институт информатики и автоматизации АН СССР. — 1988.