УДК 519.683

ДВУХУРОВНЕВАЯ ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА ДЛЯ ПОЛИНОМОВ ЛЕЖАНДРА ВЫСОКИХ ПОРЯДКОВ

В. И. Усиченко, Д. В. Заврайский

Государственное предприятие «Конструкторское бюро «Южное» имени М. К. Янгеля»

Показано доцільність дворівневої оптимізації програмного коду для обчислення лежандрових поліномів високих порядків. На основі хронометражу засобами операційної системи зроблено висновок про ефективність такої оптимізації. Данний матеріал є другою, завершаючою частиною статті [8].

Ключові слова: рівень оптимізації, ассемблер, поліноми Лежандра, потенціал, компоненти вектора гравітаційного прискорення.

The expediency of two-level optimisation of a program code for calculation Legendre polinomials of high orders is shown. The conclusion about efficacy of optimisation is made on the basis of timekeeping by operating system means. This paper is a second part of article [8].

Keywords: optimisation level, the assembler, Legendre polinomials, potential, components of a vector of gravitational acceleration.

Показана целесообразность двухуровневой оптимизации программного кода для вычисления лежандровых полиномов высоких порядков. На основе хронометража средствами операционной системы сделан вывод об эффективности такой оптимизации. Настоящий материал представляет вторую, завершающую часть статьи [8].

Ключевые слова: уровень оптимизации, ассемблер, полиномы Лежандра, потенциал, компоненты вектора гравитационного ускорения.

Цель работы, уровни оптимизации и критерии оптимальности исследуемых алгоритмов. В ходе выполнения работы ставилась задача двухуровневой оптимизации кода для алгоритмов вычисления полиномов Лежандра высоких порядков (вплоть до нескольких десятков тысяч). Затем средствами операционной системы проводилась оценка того, на сколько уменьшилось в результате оптимизации среднее время вычисления лежандрового многочлена произвольного порядка в заданной точке отрезка [-1; 1]. Такая постановка задачи обусловлена, прежде всего, высокой степенью вложенности алгоритмов вычисления полиномов в задачу разложения земного потенциала в ряд по сферическим функциям. Практическая ценность мер по оптимизации кода оценивалась на примере задачи вычисления компонент вектора гравитационного ускорения.

Основным и наиболее наглядным критерием оптимальности того или иного алгоритма является минимизация времени его машинного выполнения. В программировании проблему оптимизации принято делить на три уровня [1]:

- высокоуровневая оптимизация, основывающаяся на первоначальном выборе алгоритма. На этом этапе анализируется в первую очередь логическая структура алгоритма, число содержащихся в нем условных и безусловных переходов, циклов, оценивается время его выполнения в зависимости от объема *n* входных данных (обычно применяется оценка *O(n)*. Считается, что высокоуровневая оптимизация дает наибольший вклад в оптимизацию алгоритма в целом независимо от выбранного языка программирования;
- оптимизация среднего уровня подразумевает рациональный выбор языка программирования и эффективную реализацию алгоритма на нем. Среднеуровневая оптимизация является наиболее ответственным этапом при оптимизации алгоритма. Основные проблемы в плане оптимизации исходного кода на этом этапе чаще всего разработчику создают циклы. Именно они, как правило, являются критическими по времени выполнения участками программного кода. В связи с этим существует ряд специальных подходов к программированию циклов [1; 4];

© Усиченко В.И., Заврайский Д.В., 2018

- низкоуровневая оптимизация предполагает анализ количества тактов процессора на выполнение каждой команды и выбор рационального их порядка для конкретной архитектуры процессора. В плане низкоуровневой оптимизации наиболее эффективным считается язык ассемблера. Применение ассемблера оказывает очень сильное влияние и на другие аспекты оптимизации, например, на объем генерируемого в результате компиляции машинного кода. В силу ряда обстоятельств, обусловленных, прежде всего, особенностями программирования на ассемблере и функционирования операционной системы, было решено отказаться от низкоуровневой оптимизации алгоритмов для лежандровых полиномов. Поэтому в дальнейшем речь идет исключительно о двухуровневой оптимизации (высоко- и среднеуровневой).

Результаты двухуровневой оптимизации алгоритмов, основанных на рекуррентных соотношениях. Общеизвестно, что в силу разнообразия форм представления полиномов Лежандра существует целый ряд алгоритмов для их вычисления в заданной точке. Поэтому прежде чем приступить к высокоуровневой оптимизации нами были проанализированы четыре различных алгоритма для нахождения значений лежандровых полиномов на основе формулы Лапласа, интегральных соотношений Мейера, а также на основе применения гипергеометрической функции и рекуррентных соотношений. В результате анализа сложилось твердое убеждение, что исходный, а также оптимизированный алгоритм для вычисления полиномов Лежандра высоких порядков на [-1, 1] следует строить исходя из рекуррентного соотношения

$$P_{n+1}(z) = \frac{1}{n+1} \cdot \left[(2n+1) \cdot z \cdot P_n(z) - n \cdot P_{n-1}(z)\right] \quad (1)$$

Консольные приложения, как известно, требуют минимума ресурсов и не «отвлекают» операционную систему на «посторонние» процессы, связанные с обеспечением графического интерфейса, поэтому хронометраж вычислительного процесса будет точнее. Исходя из этого реализация и тестирование основанного на (1) алгоритма проводились в консольном режиме среды Microsoft Visual C++ 2010 для двух случаев.

В первом случае алгоритм был реализован на «чистом» C++. Соответствующий листинг мы не приводим в силу отсутствия в нем каких-либо особенностей.

Во втором случае алгоритм для (1) был реализован в том же режиме и в той же среде, но для вычислений в цикле по рекуррентной формуле (1) в C++ код был встроен ассемблерный блок. Этот вариант исходного кода на C++/MASM представлен в листинге 1 (см. приложения в конце основного текста). В обоих случаях средствами ОС проводился хронометраж времени счета.

В каждом из двух рассматриваемых случаев хронометраж велся для расчета полиномов Лежандра до стотысячного порядка. В качестве критерия быстродействия функции для вычисления по (1) использовалось усредненное время $\bar{\tau}$ вычисления полинома Лежандра произвольного порядка, которое рассчитывалось по формуле

$$\bar{\tau} = \frac{\tau_{\Sigma}}{n_{\max}}, \quad (2)$$

где $n_{\max}$ – наивысший порядок лежандрового полинома, до которого велся счет;

τ_{Σ} – суммарное время счета полиномов Лежандра от нулевого порядка до $n_{\max}$. Время τ_{Σ} определялось в миллисекундах средствами операционной системы с помощью функции *clock( )* библиотеки *time.h*, а затем (в коде на «чистом» C++) переводилось в секунды с использованием макроса *CLOCKS_PER_SEC*. Начальные результаты хронометража вычисления полиномов Лежандра $P_n(z)$ до порядка 100000 ($n_{\max} = 99999$) в случае реализации основанного на (1) алгоритма только на C++ приведены на рис. 1. Полученное время выполнения (рис. 1) для «чистого» C++ включает также время компиляции исходного кода и время вывода на экран результата для каждого из полиномов. Операция вывода на консоль – довольно медленная. Если ее исключить и оставить

вывод только для времени выполнения, то величина (2) уменьшится примерно на 15 %. С учетом этого для кода только на С++, то есть без ассемблерного блока, для усредненного времени (2) вычисления значения одного полинома Лежандра получаем:

$$\bar{\tau} \approx 0{,}85 \cdot \frac{653}{100000} \approx 0{,}0055 \quad c\,.$$

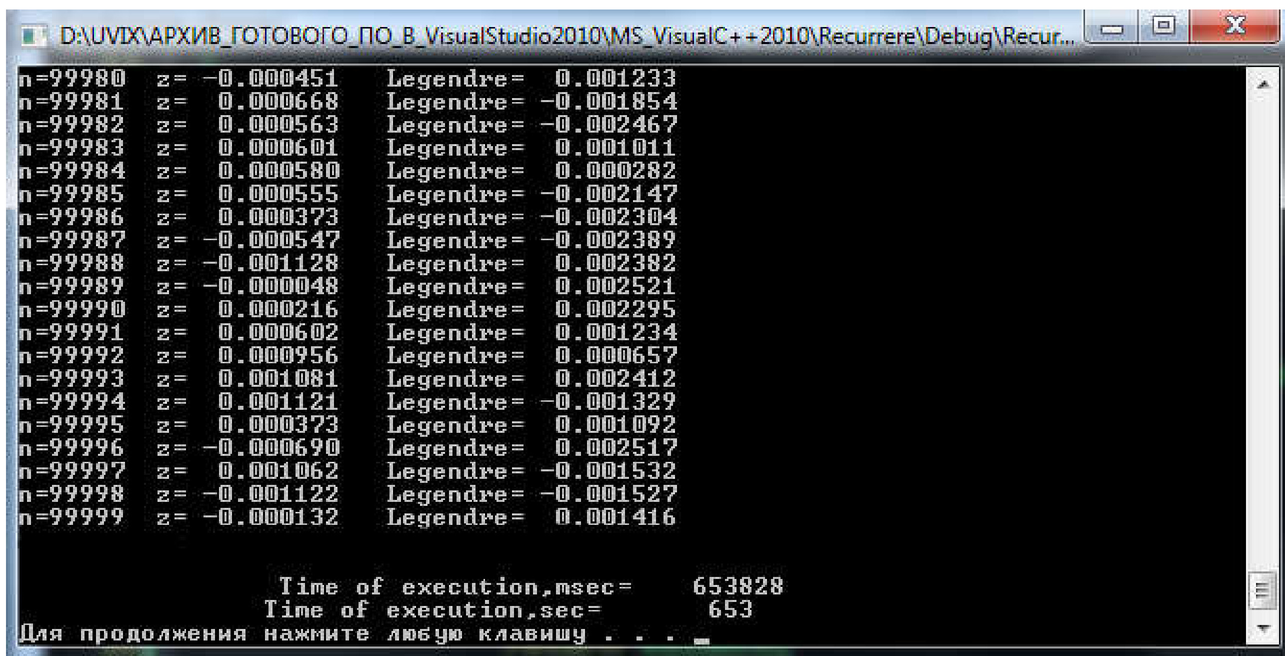

Рис. 1

При использовании блока ассемблерных команд (листинг 1 приложений) усредненное время вычисления полинома Лежандра произвольного порядка при том же n_{max} = 100000 сокращается до 0,00258 с при выводе результатов на экран в цикле, и до 0,0022 с при выводе на консоль только времени выполнения. Таким образом быстродействие алгоритма в случае ассемблерных вставок в С++ код возрастает в 0,0055/0,0022 ≈ 2,5 раза. В систематизированном виде результаты предпринятых мер по двухуровневой оптимизации кода вычисления лежандровых полиномов представлены в табл. 1.

Таблица 1

Результаты тестирования усредненного времени вычисления полиномов Лежандра высоких порядков на С++ и С++/MASM (n_{max} = 100000)

Вид хронометража	t_1, с («чистый» С ++)	t_2, с (С++/MASM)	t_1 / t_2
Хронометраж с учетом времени вывода на консоль	0,00654	0,00258	2,53
Хронометраж без учета времени вывода на консоль	0,00589	0,0022	2,68

Из анализа следует главный вывод о том, что реализация вычисления полиномов Лежандра $P_n(z)$ в цикле по n в виде ассемблерного блока сокращает общее время вычисления примерно в два с половиной раза.

Полученный результат является вполне ожидаемым. Во-первых, вычислительный цикл является наиболее критичным по времени выполнения участком программы и его построение на основе ассемблерного кода – часто применяемый и хорошо зарекомендо-

вавший себя прием. Во-вторых, использование команд сопроцессора обеспечивает использование его собственных регистров со стековой структурой и почти исключает обращение к оперативной памяти компьютера для хранения промежуточных результатов. Следовательно, при применении основанного на (1) алгоритма для лежандровых полиномов высоких порядков, предпочтительным оказывается построение соответствующей функции с использованием команд ассемблера.

В отношении *присоединенных* функций Лежандра для краткости изложения ограничимся лишь констатацией моментов, необходимых для целостного представления о результатах предпринятых мер по оптимизации кода.

Во-первых, арсенал методов построения алгоритмов вычисления присоединенных лежандровых функций несколько беднее, а сами эти методы сложнее, чем в случае основных функций.

Во-вторых, анализ свидетельствует, что предпочтение для присоединенных лежандровых функций высоких порядков и степеней также стоит отдать рекуррентным методам, основанным на двух известных соотношениях [3]:

$$P_{m,m}(z) = \sqrt{\left(1 - z^2\right)^m} \cdot \prod_{i=1}^{m}(2 \cdot i - 1) ,$$

$$P_{k,m}(z) = \frac{1}{k - m} \cdot \left[(2k - 1) z P_{k-1,m}(z) - (k + m - 1) P_{k-2,m}(z)\right] , \quad (3)$$

где второе соотношение приведено к специальному, удобному для программирования виду. И, наконец, самое главное: основанная на тех же принципах, что и в случае основных полиномов Лежандра, двухуровневая оптимизация базирующихся на (3) алгоритмов дает результаты, близкие к представленным в табл. 1.

В листинге 2 приложений представлен C++/MASM код функции *double PrisLegAsm (long order,long power, double arg)* для вычисления присоединенных функций Лежандра. Ее аргументами в порядке следования являются: порядок, степень присоединенной функции Лежандра и значение аргумента $z \in [-1, 1]$.

Необходимо признать, что степень оптимизации листинга 2 можно при необходимости повысить, если постараться уменьшить число обращений к оперативной памяти.

Проверка результатов оптимизации на примере задачи вычисления компонент вектора гравитационного ускорения. Для оценки эффективности проведенной ранее оптимизации алгоритмов вычисления полиномов Лежандра была выбрана задача расчета компонент g_x, g_y, g_z вектора гравитационного ускорения Земли в гринвичской геоцентрической системе координат с использованием полиномов Лежандра до 16 порядка включительно. Компоненты рассчитывались вдоль гринвичского меридиана в 31580 точках (шаг по широте 0,0001 рад или 0,0057 град) на геоцентрическом расстоянии 6378,137 км.

Как известно [1–3], силовую функцию гравитационного поля Земли можно представить в виде

$$U = \frac{\mu}{R}\left\{1 + \sum_{n=2}^{\infty}\sum_{m=0}^{n}\left(\frac{a_E}{R}\right)^n P_{n,m}(\sin\varphi)\right\} \cdot \left[C_{n,m}\cos(m\lambda) + S_{n,m}\sin(m\lambda)\right] , \quad (4)$$

где U – силовая функция гравитационного поля Земли;

μ, a_E – гравитационный параметр и большая полуось экваториального эллипса Земли соответственно;

R – текущее расстояние относительно притягивающего центра;

n и m – порядковые номера гармоники;

$P_{n,m}$ – главные ($m = 0$) и присоединенные ($m \neq 0$) сферические функции Лежандра;

$C_{n,m}$ и $S_{n,m}$ – зональные ($m = 0$), секторальные ($m = n$) и тессеральные ($m \neq 0$ и $m \neq n$) коэффициенты разложения гравитационного потенциала Земли в ряд для выбранной модели гравитационного поля;

λ – геоцентрическая долгота точки.

Как правило, силовую функцию (4) раскладывают на радиальную g_r, меридиональную g_φ и нормальную g_λ составляющие, которые определяются известными из теории потенциала соотношениями. Затем g_r, g_φ и g_λ с помощью соответствующей матрицы перехода пересчитываются в вектор $\{g_x, g_y, g_z\}$ правой ортогональной гринвичской геоцентрической системе координат.

Для оценки эффективности оптимизации кода вычисления полиномов Лежандра прежде всего оценивался порядок сложности алгоритма решения тестовой задачи для одной из 31580 текущих точек. Наиболее критичным по времени выполнения участком кода являются функции вычисления величин g_r, g_φ и g_λ. Подробный анализ свидетельствует, что в каждой отдельно взятой точке основные и присоединенные функции Лежандра вычисляются $\frac{3}{2}\cdot\left(n_{\max}^2+2n_{\max}-3\right)$ раз, а для N точек соответственно в N раз больше, то есть порядок сложности алгоритма вычисления величин компонент g_r, g_φ и g_λ можно принять не меньшим чем

$$O\left(\frac{3N}{2}\cdot\left(n_{\max}^2+2n_{\max}-3\right)\right), \quad (5)$$

где $n_{\max}$ – максимальный порядок используемых в расчетах полиномов Лежандра.

В нашем случае при $n_{\max} = 16$ и $N = 31580$ из (5) получим, что общие и присоединенные полиномы Лежандра, а также их производные в общей сложности будут вычислены 13500450 раз для 31580 точек.

Результаты хронометража средствами операционной системы для тестовой задачи с указанными выше начальными данными приводятся на рис. 2.

```
D:\UVIX\АРХИВ_ГОТОВОГО_ПО_B_VisualStudio2010\ASM\gComponents\Debug\gComponents.e...
g=9.7667070928216511
------------------------------
------------------------------
     Fi=89.942914679618681
Xg=-0.0095443762244112103
Yg=-0.00014908102190875494
Zg=-9.7667024575267813
g=9.7667071222189481
------------------------------
------------------------------
     Fi=89.948644257569995
Xg=-0.0085766509013431632
Yg=-0.00015496563957680508
Zg=-9.7667029517127695
g=9.7667067187435279
------------------------------
------------------------------
     Fi=89.954373835521295
Xg=-0.0075954141255341404
Yg=-0.00014731738504832992
Zg=-9.7667039659896719
g=9.7667069205181196
------------------------------
          Time of execution=544955 msec
```

Рис. 2

Как видим, полное время решения задачи составляет 545 секунд, что равняется примерно 9 минутам. Однако если учесть, что в программе применяется интенсивный вывод на консоль для каждой из 31580 точек, а сама эта операция считается одной из самых медленных, то становится понятным, что полученные результаты хронометража являются явно завышенными.

При отключении в коде вывода на консоль результатов по каждой из 31580 точек получим результаты хронометража, приведенные на рис. 3. Как видим, из общего времени решения задачи в 545 секунд на собственно вычислительное время приходится лишь 157 секунд, что составляет чуть менее 30 %.

Таким образом, среднее время расчета компонент гравитационного ускорения в одной

точке равно $\frac{157}{13500450} \approx 1{,}16 \cdot 10^{-5}$ с. В соотношениях для вычисления g_r, g_φ и g_λ по нашим оценкам на вычисление собственно полиномов Лежандра приходится около 40 % общего количества операций. Следовательно, машинное время на вычисление полиномов Лежандра в одной точке составит примерно

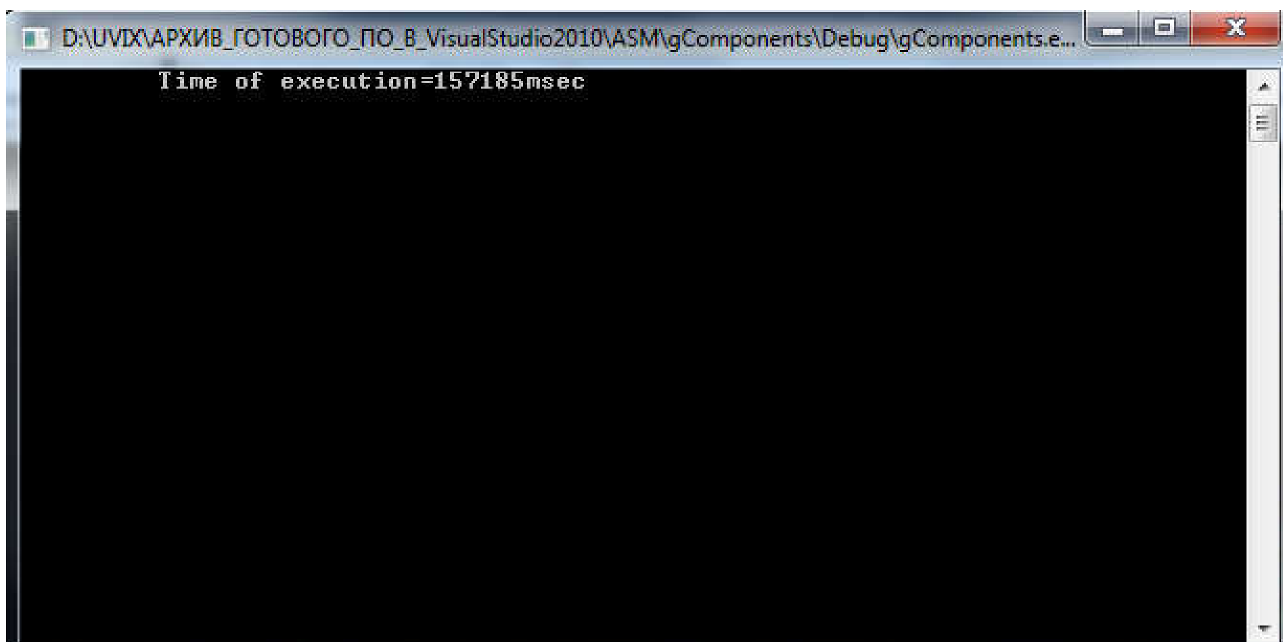

Рис. 3

$0{,}4 \cdot 1{,}16 \cdot 10^{-5} \approx 4{,}6 \cdot 10^{-6}$ с. Примерная оценка времени вычисления полиномов Лежандра и их производных в тестовой задаче в целом, то есть во всех расчетных точках, составит $13500450 \cdot 4{,}6 \cdot 10^{-6} \approx 62{,}6$ с при общем расчетном времени 157 с.

Итак, примерно 63 с машинного времени из 157 с уходит на вычисление лежандровых полиномов и их производных. Причем функции для нахождения основных и присоединенных полиномов Лежандра, имеющие наиболее глубокое вложение в код, написаны как ассемблерные блоки.

Если же учесть полученные выше результаты (табл. 1), то несложно получить, что при замене ассемблерных блоков «чистым» С++ кодом на вычисление лежандровых полиномов и их производных вместо 63 с уйдет $63 \cdot 2{,}68 = 168{,}84 \approx 169$ с.

Это приведет, в свою очередь, к увеличению общего машинного времени на ту же величину в примерно 100 с, которое достигнет уже примерно 257 с.

Отсюда несложно сделать вывод, что применение ассемблерных вставок в коде вычисления лежандровых полиномов дает приращение производительности кода для задачи вычисления компонент вектора гравитационного ускорения не менее чем на 30 % (расчетное значение 37–38 %).

Выводы.

1. Поскольку полиномы Лежандра обладают высокой степенью вложенности в код основной задачи расчета вектора гравитационного ускорения, то целесообразно исходный код их вычисления подвергнуть оптимизации высокого и среднего уровня (в идеальном случае также и низкоуровневой).

2. В результате двухуровневой оптимизации кода вычисления полиномов Лежандра можно добиться роста производительности кода основной задачи примерно на 30 %.

Приложения

Листинг 1

Исходный код для тестирования быстродействия функции вычисления полиномов Лежандра высоких порядков в среде С++ с ассемблерным модулем (двухуровневая оптимизация)

```
// RecurrereASM.cpp: определяет точку входа для консольного приложения.
```

```
//
#include "stdafx.h"
#include <conio.h>
#include <stdio.h>
#include <time.h>
       float z=0.3;  float Pk_1=1;
       int k=1; float Pk=z; float res=0;
       int k2=0; int kplus1=0;float Prom=0;
         // Код вычисления полиномов Лежандра
         double RecurrereASM (unsigned narg)
         {
                 _asm{

                    DEC narg          ; Декремент для контрольного условия в цикле
                                      ; (k<=(n-1))
                    MOV ECX,0         ; Начальное значение счетчика циклов
                          OnceMore:
                    FINIT
                    SHL k,1           ; Находим 2k в рекуррентной формуле
                    MOV EAX,k         ; Транзит 2k через регистр
                    MOV k2,EAX        ; Положили 2k в соответствующую переменную
                    INC k2            ; Нашли 2k+1
                    FLD z             ; Загружаем в стек Pk-1(z), предыдущее перед Pk
                    FLD Pk            ; Загружаем в стек Pk(z)
                    FMUL              ; Умножение z*Pk  ->  ST(0)
                    FST res           ; Запись z*Pk в переменную res
                    FILD k2           ; Загрузить 2k+1
                    FMUL              ; Нашли (2k+1)*z*Pk(z)
                    FST res           ; (2k+1)*z*Pk(z) -> res
                    SHR k,1           ; Возврат от 2k  к  k
                    FILD k            ; Подали целое k в ST(0)
                    FLD Pk_1          ; Подали Pk_1 в ST(0), k переместилось в ST(1)
                    FMUL              ; Находим k*Pk_1 -> ST(0)
                    FST Prom          ; Направили k*Pk_1 из ST(0) в переменную Prom
                    FLD res           ; Подали res в ST(0)
                    FSUB Prom  ; res - Prom = [(2k+1)*z*Pk-k*Pk-1] -> ST(0)
                    FST res           ; (res - Prom) -> res
                    INC k             ; k++
                    MOV EAX,k ;{Транзит нового значения k через регистр
                    MOV kplus1,EAX ; в переменную kplus1}
                    FLD res           ; [(2k+1)*z*Pk-k*Pk-1] -> ST(0)
                    FILD kplus1 ; Новое k -> ST(0)
                    FDIV              ; [(2k+1)*z*Pk-k*Pk-1]/(k+1) -> ST(0)
                       FST res ; Pk(z) -> res
                              ; Окончательный результат
                              ; итерации!!!
                    FLDZ              ; Очистка ST(0)
                ;======================
                    FLD Pk
                    FSTP Pk_1  ; Блок обмена местами Pk_1 и Pk
                    FLD res
                    FSTP Pk
```

```
            ;=====================
                INC ECX     ; Инкремент счетчика итераций
                CMP ECX, narg
                JB OnceMore ;Условие продолжения цикла для беззнаковых
                           ; целых (ECX НИЖЕ  narg)
                };
                     printf("n=%d", narg+1);
                     printf("  z=%f", z);
                     printf("  res=%8f\n", res);
                     return res;
        }
int _tmain(int argc, _TCHAR* argv[])
{
// Сюда вписать код для тестирования функции double RecurrereASM (unsigned narg) в
//соответствии со своей методикой ( native C++)
Printf ("Time of execution=%10d\n",clock()); // Хронометраж средствами ОС в мсек
    _getch(); // Ожидание нажатия клавиши для закрытия окна
    return 0;
}
```

Листинг 2

Исходный C++/MASM код для вычисления присоединенных функций Лежандра по заданным значениям аргумента, порядка и степени (двухуровневая оптимизация)

```
// PrisLegendreASM.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include "stdio.h"
#include "conio.h"
#include "math.h"
//Вычисление присоединенной функции Лежандра
double PrisLegAsm (long order,long power, double arg)
{
        long k(0); double Pnm(0); double Pmm(1);
        double Pk_1 = 0; double Pk_2 = 0;
        long kPlusm_1,Numb2k_1 = 0; long k_m; long checkLoop(0);
        double P1,P2,P1_P2 = 0;//NUMB2k_1 = 2*k-1
        long indx(1);
        if(arg =1)
        {
                Pnm = 0;
        }
        else if (order = 0)
        {
          Pnm = 0;
        }
    else if (order < power)
        {
          Pnm = 0;
        }
        else
        {
                k = power;
```

```
// Находим Pmm
   _asm
   {                                ; Вычисляем цикл для Pmm
       MOV ECX,1            ; Выставили счетчик числа циклов для Pmm
     NewAction:
           SHL indx,1           ; 2*i
           DEC indx             ; 2*i-1
                      FINIT
           FILD indx                  ; (2*i-1) -> ST(0)
           FLD Pmm                      ; Pmm -> ST(0), а (2*i-1) -> ST(1)
           FMUL                          ; Pmm*=(2*i-1)
           FSTP Pmm                    ; Обновление значения Pmm
           INC ECX                   ; Увеличили счетчик цикла на 1
           MOV indx, ECX           ; Обновляем индекс
           CMP ECX, power
           JLE NewAction
   };    // Конец цикла для Pmm (первое уравнение из (3))
          //Окончательное значение Pmm
           Pmm* = sqrt(pow((1-arg*arg),power));            //+
                      Pk_1 = Pmm;Pk_2 = 0;
          _asm
                    {
   ;Установка счетчика цикла с учетом одной итерации по Pmm
                        MOV EAX, order
                        MOV EBX, power
                        SUB EAX, EBX
                        MOV ECX, EAX
        Action2:
                             FINIT
                              INC k      ; k++
                                         ; Вычисляем P2
                              MOV EA X, k
                              MOV EBX, power
                              ADD EAX, EBX
                              DEC EAX
                              XCHG kPlusm_1, EAX
                              FLD kPlusm_1  ; (k+m-1) ->  ST(0)
                              FLD Pk_2         ; Pk_2 -> ST(0), kPlusm_1 -> ST(1)
                              FMUL              ; (k+m-1)*Pk_2 -> ST(0)
                              FSTP P2            ; (k+m-1)*Pk_2 -> ~P2
                                                ; Вычисляем P1
                        MOV EAX, k
                              SHL EAX,1                   ; 2k
                              DEC EAX                     ; (2k - 1)
                              XCHG Numb2k_1, EAX
                              FLD arg                          ; z -> ST(0)
                              FLD Pk_1                         ; Pk_1 -> ST(0)
                              FMUL           ; z*Pk_1 -> ST(0)
                              FSTP P          ; z*Pk_1 -> P1 (временное значение P1)
                              FLD Numb2k_1    ; (2k-1) -> ST(0)
                              FLD P1                ; P1 -> ST(0), (2k-1) -> ST(1)
```

```
            FMUL                  ; (2k-1)*z*Pk_1 -> ST(0)
            FSTP P1               ; ST(0)=(2k-1)*z*Pk_1 -> ~P1
                                  ; Конец вычисления P1
        FLD P1            ; P1 -> ST(0)
            FSUB P2               ; P1-P2 = (2k-1)*z*Pk_1- (k+m
                                      ; -1)*Pk_2 -> ST(0)
            FSTP P1_P2                    ; P1-P2 -> ~
            FLD power                     ; z -> ST(0)
            FLD k                         ; k -> ST(0), z -> ST(1)
            FSUB power                    ; (k-m) -> ST(0)
            FSTP k_m                      ; (k-m) -> ~
            FLD P1_P2                     ; (P1-P2) -> ST(0)
            FLD k_m               ; (k-m) -> ST(0), P1-P2->ST(1)
            FDIV        ; Нашли Pnm в цикле. Pnm -> ST(0)
            FSTP Pnm              ; Pnm -> ~
                                ; Конец одной итерации
        FLD Pk_1              ; Pk_1 -> ST(0)
            FSTP Pk_2             ; ST(0) = Pk_1 -> Pk_2
            FLD Pnm               ; Pnm -> ST(0)
            FSTP Pk_1             ; ST(0)=Pnm -> Pk_1
            DEC ECX
            CMP ECX, 0    ; Контроль управляющей ~
            JG Action2
      }; // End _asm
            if (order==power)
            {
        Pnm = Pmm;
            }
            else
            {
                ;
            }
  }
        return Pnm;
}
```

Библиографические ссылки

1. Зубков С.В. Assembler. Язык неограниченных возможностей. Москва: ДМК Пресс, 1999. 614 с.

2. Дубошин Г.Н. Небесная механика. Основные задачи и методы. Москва : Наука, 1968. 799 с.

3. Бордовицына Т.В., Авдюшев В.А. Теория движения искусственных спутников Земли. Аналитические и численные методы. Томск : Изд-во Томского ун-та, 2007. 175 с.

4. Пирогов Владислав. Ассемблер для Windows. Санкт-Петербург : БХВ-Петербург, 2007. 895 с.

5. Хортрон Айвор. Visual C++2010. Полный курс. Москва – Санкт-Петербург – Киев : Диалектика, 2011. 1206 с.

6. Джосаттис Николаи М. Стандартная библиотека С++. Справочное руководство. Москва – Санкт-Петербург – Киев : Диалектика, 2014. 1129 с.

7. Шилдт Герберт. Справочник программиста по С/С++. Москва – Санкт-Петербург – Киев : Вильямс, 2003. 344 с.

8. Усиченко В.И., Крюков А.В. Анализ особенностей некоторых алгоритмов вычисления лежандровых полиномов высоких порядков. *Системне проектування та аналіз характеристик аерокосмічної техніки*: зб. наук. праць. Т. XXIII. Дніпро : Ліра, 2017.

Надійшла до редколегії 08.01.2018