

С/к Введение в современное программирование (v.5.5)

Физфак МГУ. 2006/7 уч. год.

Лекция 24

Построение циклов, инварианты циклов -- WRAP-UP LQ-разложение как иллюстрация:

- полезности понятия инварианта цикла;
 - важности четкой связи математика-программа;
 - вреда рукосудных "оптимизаций".
- LQ-разложение очень важно само по себе (линеаризация -- один из ключевых матем. приемов).

Построение циклов, инварианты циклов -- WRAP-UP

Вспомним все основные циклы, которые уже встретились:

Полный проход:

```

22  i := 0; s := 0;
23  WHILE i < LEN( a ) DO
24      s := s + a[ i ];
25      i := i + 1;
26  END;

28  Охрана = i < LEN( a )
29  Инвариант = ( s = сумма всех a[ j ] с j < i )
30  На выходе ~Охрана & Инвариант => s = сумма всех вообще a[ j ].

```

Линейный поиск:

```

34  i := 0;
35  WHILE ( i < LEN( a ) ) & ~( a[ i ] = x ) DO
36      i := i + 1
37  END;
38  IF i < LEN( a ) THEN
39      ASSERT( a[ i ] = x ); (* нашли, где x *)
40  ELSE
41      ASSERT( i = LEN( a ) ); (* значения x в массиве нет *)
42  END;

44  Охрана = ( условие ограничения поиска ) & ~( условие поиска ).
45  Инвариант = ( среди элементов a[ j ] с j < i значения x нет )
46  На выходе: либо i указывает на первое от начала появление x,
47  либо i указывает за пределы массива, т.е. значения x там нет.

```

Умножение (лекция 13):

```

50
51  x := x0; y := y0; s := 0;
52  ASSERT( ( x >= 0 ) & ( y >= 0 ) );
53  WHILE x > 0 DO (* охрана *)
54      ASSERT( x*y + s = x0*y0 ); (* инвариант *)
55      IF ODD( x ) THEN
56          s := s + y
57      END;
58      y := ASH( y, 1 ); (* y*2 *)
59      x := ASH( x, -1 ); (* x DIV 2 *)
60      ASSERT( x*y + s = x0*y0 ); (* инвариант *)
61  END;
62  ASSERT( ( x*y + s = x0*y0 ) & ( x = 0 ) );
63  (* значит: *)
64  ASSERT( s = x0 * y0 );

```

Слияние (лекция 23, процедура Merge модуля Kurs2006MergeSort)

```

68  (* основной цикл: *)
69  WHILE ( e0 # NIL ) & ( e1 # NIL ) DO
70      IF e0.n < e1.n THEN
71          t := e0; e0 := s0.Next();
72      ELSE
73          t := e1; e1 := s1.Next();
74      END;
75      wr.Put( t );
76  END;

78  (* инвариант:
79      wr "построил" все элементы до e0 в s0 и до e1 в s1
80      & все элементы, оставшиеся в s0 и s1, не младше всех, что уже
81      отправлены в wr. *)

82
83  (* завершение: *)
84  WHILE e0 # NIL DO
85      t := e0; e0 := s0.Next();
86      wr.Put( t );
87  END;
88  WHILE e1 # NIL DO
89      t := e1; e1 := s1.Next();
90      wr.Put( t );
91  END;
92

```

Цикл из quicksort (лекция 22):

```

(* i и j движутся навстречу с краев,
   оставляя за собой эл-ты, соответственно, <= и => pivot
*)
(* три сегмента: i0 -- i; i -- j; j -- j0 *)
WHILE i < j DO
  IF a[i] < pivot THEN
    INC(i)
  ELSIF a[j-1] > pivot THEN
    DEC(j)
  ELSE
    t := a[i]; a[i] := a[j-1]; a[j-1] := t;
    INC(i); DEC(j)
  END
END;
ASSERT( (i > i0) & (i < j0), 101 );

```

Инвариант:

все элементы слева от i <= "опорного значения" (pivot)
 & все элементы справа от j => pivot

Во всех случаях есть

инвариант, который выполняется в начале и в конце тела цикла на каждом шаге -- следовательно, и непосредственно перед началом и после окончания цикла.

Обычно также есть

"бегунок" с ограничением (кроме умножения).

На каждом шаге рассматривается одна или больше альтернатив и в каждом случае предпринимаются какие-то действия, гарантирующие приближение "бегунка" к ограничению.

Общий цикл Дейкстры:

```

LOOP
  ASSERT( invariant );
  IF охрана-0 THEN
    продвижение-0

  ELSIF охрана-1 THEN
    продвижение-1

  ELSIF ...
  ...
  ELSE
    EXIT
  END;
  ASSERT( invariant );
END;

```

Все примеры -- частный случай сего. (**Упр*** Убедиться.)

См. рекомендованные книжки Э.Дейкстры (Дисциплина прог-я) и Д.Гриса (Наука прог-я).

Практическая рекомендация:**базовый уровень:**

разобравшись, выучить приведенные образцы циклов ("99.5%" всех случаев)

advanced level:

понять структуру цикла Дейкстры (по примерам, в т.ч. из книжек), чтобы уметь удостоверяться в корректности строимых циклов -- обычно в трудных случаях достаточно:

- строить цикл по схеме Дейкстры, при этом:
- обязательно сформулировать инвариант
- добавлять ветви до тех пор, пока логика не гарантирует исчерпание всех возможностей
- в каждой ветви гарантировать нетривиальное продвижение к цели
- в каждой ветви гарантировать восстановление инварианта
- убедиться, что всего нужно конечное число шагов

professional level:

уметь "выводить" циклы по Дейкстре.

NB Очень много научного софта написано людьми остроумными, но имеющим никакое понятие о "культурном" прог-и.

В частности, "оптимизации" рукосудным методом --> нечитабельные, **неконтролируемые** циклы.

Неудержимое стремление "оптимизировать" идет от 1950-х гг., когда было важно сэкономить несколько байт, несколько обращений к массиву и т.п. еще до первого запуска программы. И когда прогон подобной процедуры был единственным пунктом задачи.

Что делать в таких случаях? Если нужен полный контроль -- переписать, отталкиваясь от матем. формул, поглядывая на старый код.

Пример:**LQ-разложение**

Обобщение метода Гаусса; лучший метод решения лин. ур-й, вычисления детерминантов, обращения матриц.
 Фундаментальная задача, т.к. прием линеаризации применяется постоянно.
 Нужен абсолютный контроль.
 Первое же обращение к Numerical Recipes in ... -- именно по этому поводу.

Что видим? (8 страничек распечатаны отдельно)

Видим весь букет:

0) пописали формулы

1) бросились программировать

2) ни о каком инварианте цикла речи нет --> цикл уродливый

3) по ходу, до прояснения структуры программы -- рукосудные оптимизации ("our implementation has one additional wrinkle")

4) уже на шаге 1 про формулы забыто.

В результате:

— нечитабельный "оптимизированный" код, который трудно верифицировать и невозможно модифицировать (например, чтобы реализовать стабилизирующую

198 перестановку столбцов в добавок к реализованной перестановке строк, хотя бы
 199 для проверки);
 200 — утверждения авторов о свойствах (напр., избыточность перестановки
 201 столбцов) невозможно проверить с этим кодом -- а у нас мало ли какая особая
 202 ситуация;
 203 — делается вывод о невозможности "эффективной" реализации перестановки
 204 столбцов, справедливый только для узкого архаического понятия
 205 "эффективности", принятого авторами (в реале многоуровневые механизмы
 206 кэширования и упреждающих вычислений современных процессоров
 207 обесмысливают мелочную оптимизацию, а общий объем и сложность
 208 программных комплексов низводят проблему до уровня нескольких процентов).
 209

210 *Качество/корректность решения предшествует "оптимизациям"*

211
 212 **Ниже даем**

213 Кусок документации и код реализации LQ-разложения в окультуренном виде
 214 (библиотека Numath, автор Ф.Ткачев, для эксперимента Troitsk-V-mass).
 215 Сделано по матем. формулам "с нуля".
 216 Оптимизации (пресловутые перестановки строк и столбцов) выполнены не
 217 рукоусойно, а с опорой на формулы (явное введение перестановок).
 218 Явно понят и сформулирован инвариант цикла, в программе максимально чисто
 219 реализуется именно цикл по этому инварианту, никакого дополнительного
 220 рукоусойства не делается -- в результате оказывается открытой дорога к той
 221 оптимизации, которую было "невозможно эффективно реализовать".
 222

223 **NB** Здесь важный и общий момент: необходимость **возвращаться** к формулам,
 224 чтобы решить возникающие при реализации проблемы еще на математическом
 225 уровне (где это сделать зачастую гораздо легче), и потом снова делать код
 226

227 ИТЕРАТИВНОСТЬ РАЗРАБОТКИ (возвращаться и заново переделывать,
 228 начиная с предыдущих уровней абстракции, а не просто латать, оставаясь на
 229 текущем)

230
 231 ВКЛЮЧЕННОСТЬ МАТЕМАТИЧЕСКОГО УРОВНЯ В ПРОЦЕСС РАЗРАБОТКИ **на**
 232 **равной ноге**

233
 234 **Упр** Изучить и сравнить два варианта.
 235

236 **(Numath)Matrices**

237
 238 **LU-разложение**
 239

240 Пусть есть матрица $a_{i,j}$ (у нас все индексы меняются от 0 до dim - 1).

241 Конечная цель -- научиться решать несколько родственных между собою задач:
 242 вычислять определитель матрицы а, обращать ее, а также решать уравнение
 243 (возможно, многократно):

$$244 \sum_j a_{i,j} x_j = y_i \quad (0)$$

245 Все эти задачи удобно решаются, если найти представление матрицы а в виде
 246 произведения нижней (lower) и верхней (upper) треугольных матриц (формулы
 247 даны ниже).

248
 249 Поскольку алгоритмы особо не зависят от конкретного порядка индексов i, j,
 250 этим можно воспользоваться для повышения численной устойчивости алгоритма.
 251 Поэтому будем искать LU-разложение для некоторой матрицы А, некоторым
 252 образом связанной с исходной матрицей а:

$A_{i,j} = \sum_k L_{i,k} U_{k,j} = \sum_{k \leq \min(i,j)} L_{i,k} U_{k,j} \quad (*)$

254 где L, U -- нижняя и верхняя треугольные матрицы (что и показано во второй
 255 сумме посредством предела суммирования по k). У одной из матриц значения на
 256 диагонали можно положить равными 1, например, у L: $L_{n,n} = 1$ для всех n.

257 Будем ссылаться на отдельные уравнения системы (*) по паре i,j в соответствии
 258 с левой частью.

259
 260 Связь между а и А не должна быть сложной (по возможности). Будем
 261 использовать такую общую форму связи:

$A_{i,j} = \bar{a}_{\pi(i),\pi'(j)},$

263 где две перестановки $\pi(i), \pi'(j)$ должны обеспечить наличие по возможности
 264 больших (по абс. величине) элементов на диагонали у А, а матрица \bar{a} связана
 265 с исходной сл. образом:

$\delta_i \bar{a}_{i,j} = a_{i,j}, \text{ где } \delta_i = \max_j |a_{i,j}| \quad ('abar')$

267 Тогда если переписать исходные уравнения в этих терминах,

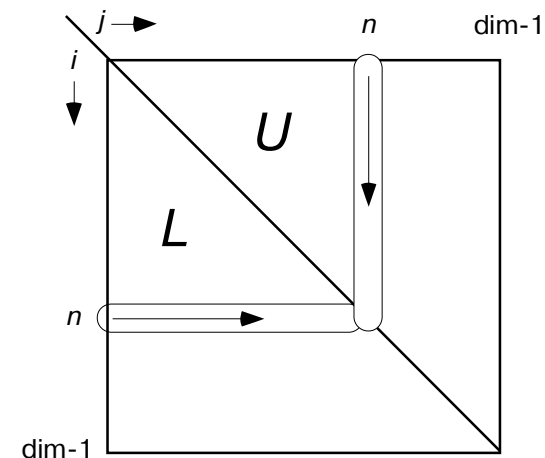
$\sum_j \bar{a}_{i,j} x_j = \delta_i^{-1} y_i \quad (00)$

269 то максимальный коэффициент в каждом уравнении равен 1. Именно такая
 270 нормировка лучше всего подходит для выбора "больших" коэффициентов.
 271

272 **Модифицированный Crout-алгоритм**

273 Crout заметил, что систему ур-й (*) можно решить за один специальный обход
 274 по i, j, начиная с угла i=0, j=0 и расширяя квадрат, добавляя колонку справа и
 275 ряд снизу (см. картинку).

276
 277 **Замечание.**
 278 Первоначальный Crout-
 279 алгоритм, как он описан в
 280 *Numerical Recipes...*, вместо
 281 прохода по горизонтали,
 282 продолжает идти по
 283 колонке до самого низа. Это
 284 тоже формально корректно,
 285 но такая организация
 286 циклов нарушает
 287 симметрию в задаче и
 288 поэтому путает структуру
 289 алгоритма — без реальной
 290 нужды, но с неприятными
 291 последствиями.
 292



Итак, пусть n пробегает от 0 до $\dim - 1$, и пусть все элементы матриц L и U , у которых $i < n$ и $j < n$, уже найдены.

Тогда сначала будем находить $L_{n,j}$ проходом по j от 0 до $n - 1$ из уравнений

$$A_{n,j} = \sum_{k \leq j} L_{n,k} U_{k,j} = \sum_{k < j} L_{n,k} U_{k,j} + L_{n,j} U_{j,j}$$

откуда

$$L_{n,j} = \left(A_{n,j} - \sum_{k < j} L_{n,k} U_{k,j} \right) / U_{j,j} \quad (***)$$

Видно, что если идти по j от 0 по возрастанию, и тогда на каждом шаге используются только уже вычисленные величины.

Затем аналогичным образом найдем $U_{i,n}$ для i от 0 до n из соответствующих уравнений:

$$A_{i,n} = \sum_{k \leq i} L_{i,k} U_{k,n} = \sum_{k < i} L_{i,k} U_{k,n} + U_{i,n}$$

(Помним, что по нашим соглашениям $L_{n,n} = 1$.)

Получим:

$$U_{i,n} = A_{i,n} - \sum_{k < i} L_{i,k} U_{k,n} \quad (**)$$

И здесь по i надо идти от 0 по возрастанию, тогда на каждом шаге используются только уже вычисленные величины.

110
111

Решение уравнения

Пусть мы нашли L , U как описано выше. Теперь будем решать систему уравнений (00) (эквивалентную исходной системе (0)). Перепишем (00), тождественно заменив индексы так:

$$\sum_j \bar{a}_{\pi(i), \pi'(j)} x_{\pi'(j)} = y_{\pi(i)} / \delta_{\pi(i)}$$

В суммировании по j изменился лишь порядок суммирования, поскольку $\pi(j)$ — перестановка.

Наконец, правую часть можно заменить на A , и система примет вид:

$$\sum_j A_{i,j} \bar{x}_j = y_{\pi(i)} / \delta_{\pi(i)} \equiv \bar{y}_i, \quad \bar{x}_j \equiv x_{\pi'(j)}$$

В матричном виде $\bar{y} = A\bar{x} = LU\bar{x}$. Поэтому можно решать систему в два шага: сначала решим систему $\bar{y} = Lz$, затем $z = U\bar{x}$. Первая система расписывается так:

$$\bar{y}_i = \sum_{i \geq k} L_{i,k} z_k = \sum_{i > k} L_{i,k} z_k + L_{i,i} z_i \quad (L)$$

откуда алгоритм решения:

$$z_i = \bar{y}_i - \sum_{i > k} L_{i,k} z_k \quad (L1)$$

где идти надо по возрастанию i .

Вторая система расписывается так:

$$z_k = \sum_{j \geq k} U_{k,j} \bar{x}_j = \sum_{j > k} U_{k,j} \bar{x}_j + U_{k,k} \bar{x}_k \quad (U)$$

откуда алгоритм решения:

$$\bar{x}_k = \left(z_k - \sum_{j > k} U_{k,j} \bar{x}_j \right) / U_{k,k} \quad (U1)$$

а идти надо по убыванию k .

334
335

Вычисление определителя

337

Пусть мы нашли L , U как описано выше. Тогда для определителя имеем:

$$\det a = \varepsilon^\pi \varepsilon^{\pi'} \prod_n \delta_n \times \det L \times \det U = \varepsilon^\pi \varepsilon^{\pi'} \prod_n \delta_n \times \prod_n U_{n,n}$$

где ε^π и $\varepsilon^{\pi'}$ — четности двух перестановок.

341
342

Оптимизации

344

— Матрицы U и L можно хранить в одной матрице (нужно только в формулах заранее учесть, что для диагональных элементов $L_{n,n} = 1$ -- как мы и

поступили). Более того, для хранения U и L можно использовать саму же матрицу A , т.к. каждый ее элемент используется ровно один раз для вычисления соотв. элемента L или U , и больше не нужен.

На практике можно просто использовать три указателя A , L , U на одну и ту же матрицу.

352

— *Numerical recipes*... предпочитают не делать деление на дельту в явном виде. Все формулы можно преобразовать соотв. образом.

355

— После вычисления матрицы A , матрица \bar{a} больше не нужна. Поэтому возникает мысль не использовать отдельную матрицу \bar{a} . Однако чтобы это не интерферировало с хранением L , U в том же массиве, нужно в явном виде сделать перестановки элементов, превращающие \bar{a} в A . Заметим, что в нашей модификации *CROUT*-алгоритма эти перестановки не перепутываются с проходом по столбцу, как в первоначальном *CROUT*-алгоритме. Поэтому мы можем сделать все перестановки (включая перестановки столбцов) в явном виде, причем заранее.

364

— Можно отказаться, например, от перестановки столбцов. Какова плата качеством решения за это?

Вот усредненные ошибки, полученные на случайных 3-мерных матрицах (10М штук; каждый элемент матрицы есть независимое случайное число из интервала $(0,1)$):

370

без перестановок:

$$\text{average error} = 4.811626424894269E-12$$

373

только строки:

$$\text{average error} = 1.008826629616447E-12$$

улучшение на фактор ~ 5

377

строки и столбцы:

$$\text{average error} = 7.348820730973621E-13$$

```

380 улучшение на фактор ~1.4
381
382 NB На некоторых матрицах может иметь место ухудшение точности по
383 сравнению с перестановкой только строк, хотя в среднем имеет место
384 улучшение результатов.
385
386 Ниже дается лишь часть модуля, соответствующая реализации LU-разложению.
387 Исключены:
388   процедуры-фасады (прячущие "кухню" LU-разложения для типичных
389   ситуаций);
390   простые реализации для простых частных случаев (dim=2, dim=3), которые
391   в жизни встречаются часто, а делать для них полное LU-разложение -- дорого.
392
393 DEFINITION NumathMatrices;
394
395   CONST
396     altPivot = 2;
397     noPivot = 0;
398     stdPivot = 1;
399
400   TYPE
401     Directory = POINTER TO ABSTRACT RECORD
402       (dir: Directory) NewSolver (IN a: Matrix; dim, pivotType:
403   INTEGER; old: Solver): Solver, NEW, ABSTRACT
404     END;
405
406     Solver = POINTER TO ABSTRACT RECORD
407       (s: Solver) Det (): REAL, NEW, ABSTRACT;
408       (s: Solver) Invert (OUT inv: Matrix), NEW, ABSTRACT;
409       (s: Solver) Solve (IN y: Vector; OUT x: Vector), NEW,
410   ABSTRACT
411     END;
412
413     Matrix = ARRAY OF Vector;
414
415     Vector = ARRAY OF REAL;
416
417   VAR
418     dir-: Directory;
419     stdDir-: Directory;
420
421   PROCEDURE DistFromUnit (IN c: Matrix): REAL;
422   PROCEDURE SetDir (d: Directory);
423
424 END NumathMatrices.

```

```

423 MODULE NumathMatrices; (* 15 мар 2007 г. (0250) by FVT *)
424 IMPORT Math;
425
426 CONST noPivot* = 0; stdPivot* = 1; altPivot* = 2;
427
428 TYPE
429   Vector* = ARRAY OF REAL;
430   Matrix* = ARRAY OF Vector;
431
432   Directory* = POINTER TO ABSTRACT RECORD END;
433
434   Solver* = POINTER TO ABSTRACT RECORD END;
435
436   StdDirectory = POINTER TO RECORD ( Directory ) END;
437   LU = POINTER TO LIMITED RECORD ( Solver )
438     dim, sign: INTEGER;
439     abar, A, L, U: POINTER TO Matrix;
440     amax, tmp: POINTER TO Vector;
441     pi, pj: POINTER TO ARRAY OF INTEGER; (* transpositions *)
442     pivotType: INTEGER;
443     ok: BOOLEAN
444   END;
445
446 VAR dir-, stdDir-: Directory;
447   lastAborted: LU;
448
449 PROCEDURE ( dir: Directory ) NewSolver* ( IN a: Matrix; dim,
450 pivotType: INTEGER; old: Solver ): Solver, NEW, ABSTRACT;
451
452 PROCEDURE ( s: Solver ) Det* (): REAL, NEW, ABSTRACT;
453 PROCEDURE ( s: Solver ) Solve* ( IN y: Vector; OUT x: Vector ), NEW,
454 ABSTRACT;
455 PROCEDURE ( s: Solver ) Invert* ( OUT inv: Matrix ), NEW, ABSTRACT;
456
457 (** LU-разложение
458 *****)
459
460 PROCEDURE Normalize ( dim: INTEGER; IN a: Matrix; OUT amax:
461 Vector; OUT abar: Matrix; OUT ok: BOOLEAN );
462 (* all arrays may be longer than dim *)
463   VAR i, j: INTEGER; max: REAL;
464 BEGIN
465   ok := TRUE;
466
467   FOR i := 0 TO dim - 1 DO
468     max := 0;
469     FOR j := 0 TO dim - 1 DO
470       max := MAX( max, ABS( a[ i, j ] ) )
471     END;
472     amax[ i ] := max;

```

```

473     IF max = 0 THEN ok := FALSE; RETURN END
474 END;
475
476 FOR i := 0 TO dim - 1 DO
477     FOR j := 0 TO dim - 1 DO
478         abar[ i, j ] := a[ i, j ] / amax[ i ]
479     END
480 END;
481 RETURN
482 END Normalize;
483
484 PROCEDURE EvalA ( dim, ptype : INTEGER; IN abar: Matrix; OUT A:
485 Matrix; OUT pi, pj: ARRAY OF INTEGER; OUT sign: INTEGER );
486     (* all arrays may be longer than dim *)
487     VAR i, i0, i0max, j, j0, j0max, n, t: INTEGER; max, next: REAL;
488     BEGIN
489         (* The numbers of rows and columns in the original matrix are called
490         "absolute".
491         They are stored in the arrays pi, pj, perhaps in a different order. *)
492
493         (* unit permutation: *)
494         FOR n := 0 TO dim - 1 DO
495             pi[ n ] := n; pj[ n ] := n
496         END;
497         sign := +1;
498
499         (* depending on the chosen type, the pivoting is done,
500         i.e. a reordering of rows and/or columns so as to large elements are on
501         the diagonal: *)
502         CASE ptype OF
503             | noPivot: (* pivoting is not attempted *)
504
505             | stdPivot:
506                 (* This version of pivoting is similar to Numerical Recipes: the largest
507                 element is selected from the current column only, from the remaining rows. The
508                 implementation is slightly different, however. *)
509                 (* Construct the nXn submatrix by increasing n: *)
510                 FOR n := 0 TO dim - 1 DO
511                     (* Loop invariant: each mXm submatrix with m<n has the desired
512                     properties. *)
513                     (* Look at the remaining rows in the column j = n and find the largest
514                     element: *)
515                     max := 0;
516                     FOR i0 := n TO dim - 1 DO
517                         i := pi[ i0 ]; (* i enumerates unprocessed rows, but in a hard-to-
518                         predict order that depends on what transpositions were done for preceeding n; note
519                         that i is an "absolute" row number *)
520                         next := ABS( abar[ i, n ] );
521                         IF max < next THEN
522                             max := next;
523                             i0max := i0;
524                     END
525                     END;

```

```

526
527         (* remember the found row in pi; the actual moving of the elements
528         will be done separately: *)
529         (* ASSERT( ( i0max # n ) = ( pi[ i0max ] # pi[ n ] ), 103 ); *)
530         IF i0max # n THEN
531             t := pi[ n ]; pi[ n ] := pi[ i0max ]; pi[ i0max ] := t;
532             sign := - sign;
533         END;
534     END;
535
536     | altPivot:
537         (* Similar to the above, only the largest element is selected from the
538         entire remaining submatrix, and both rows and columns are transposed.
539         *)
540
541     FOR n := 0 TO dim - 1 DO
542         max := 0;
543         FOR i0 := n TO dim - 1 DO
544             i := pi[ i0 ];
545             FOR j0 := n TO dim - 1 DO
546                 j := pj[ j0 ]; (* similar to i, only for columns *)
547                 next := ABS( abar[ i, j ] );
548                 IF max < next THEN
549                     max := next;
550                     i0max := i0; j0max := j0;
551             END
552         END
553     END;
554
555     (* ASSERT( ( i0max # n ) = ( pi[ i0max ] # pi[ n ] ), 103 ); *)
556     IF i0max # n THEN
557         t := pi[ n ]; pi[ n ] := pi[ i0max ]; pi[ i0max ] := t;
558         sign := - sign;
559     END;
560
561     (* ASSERT( ( j0max # n ) = ( pj[ j0max ] # pj[ n ] ), 104 ); *)
562     IF j0max # n THEN
563         t := pj[ n ]; pj[ n ] := pj[ j0max ]; pj[ j0max ] := t;
564         sign := - sign;
565     END
566 END;
567
568 END; (* CASE pType *)
569
570     (* the actual transposition of the matrix elements: *)
571     FOR i := 0 TO dim - 1 DO
572         FOR j := 0 TO dim - 1 DO
573             A[ i, j ] := abar[ pi[ i ], pj[ j ] ]
574         END
575     END;
576     (* an interesting (and doable) problem is how to accomplish this within the
577     same matrix. *)
578 END EvalA;

```

```

579
580  PROCEDURE EvalLU ( dim: INTEGER; IN A: Matrix; VAR L, U: Matrix;
581  OUT ok: BOOLEAN );
582  VAR i, j, k, n: INTEGER; s: REAL;
583  BEGIN
584  ok := TRUE;
585  FOR n := 0 TO dim - 1 DO
586  FOR j := 0 TO n - 1 DO
587  (* evaluation of the formula (**): *)
588  s := A[ n, j ];
589  FOR k := 0 TO j - 1 DO
590  s := s - L[ n, k ] * U[ k, j ]
591  END;
592  L[ n, j ] := s / U[ j, j ]
593  END;
594
595  FOR i := 0 TO n DO
596  (* evaluation of the formula (**): *)
597  s := A[ i, n ];
598  FOR k := 0 TO i - 1 DO
599  s := s - L[ i, k ] * U[ k, n ]
600  END;
601  U[ i, n ] := s
602  END;
603  IF U[ n, n ] = 0 THEN ok := FALSE; RETURN END;
604  END;
605  END EvalLU;
606

```

```

607  PROCEDURE ( dir: StdDirectory ) NewSolver ( IN a: Matrix; dim,
608  pivotType: INTEGER; old: Solver ): LU;
609  VAR lu: LU; i, j, n: INTEGER; max: REAL;
610  BEGIN
611  ASSERT( ( dim > 0 ) & ( LEN( a, 0 ) >= dim ) & ( LEN( a, 1 ) >= dim ), 20 );
612  IF old # NIL THEN
613  WITH old: LU DO
614  lu := old
615  ELSE
616  END;
617  END;
618  IF lu = NIL THEN lu := lastAborted; lastAborted := NIL END;
619  IF lu = NIL THEN NEW( lu ) END;
620
621  lu.ok := FALSE;
622  CASE pivotType OF
623  | noPivot, stdPivot, altPivot:
624  lu.pivotType := pivotType
625  ELSE
626  HALT( 21 )
627  END;
628
629  IF dim > lu.dim THEN
630  NEW( lu.amax, dim );
631  NEW( lu.tmp, dim );
632  NEW( lu.abar, dim, dim );
633  NEW( lu.A, dim, dim );
634  lu.L := lu.A; lu.U := lu.A; (* reuse of same array possible thanks to
635  algorithm used in EvalLU *)
636  (* NEW( lu.L, dim, dim ); NEW( lu.U, dim, dim ); *)
637  NEW( lu.pi, dim ); NEW( lu.pj, dim );
638  END;
639  lu.dim := dim;
640
641  (* сделать нормировку -- вычислить delta, abar *)
642  Normalize( dim, a, lu.amax, lu.abar, lu.ok );
643
644  IF lu.ok THEN
645  EvalA( dim, lu.pivotType, lu.abar, lu.A, lu.pi, lu.pj, lu.sign );
646  EvalLU( dim, lu.A, lu.L, lu.U, lu.ok );
647  END;
648
649  IF ~lu.ok THEN lastAborted := lu; lu := NIL END;
650  RETURN lu
651  END NewSolver;
652

```

```

653  PROCEDURE ( lu: LU ) Det (): REAL;
654      VAR res: REAL; n: INTEGER;
655  BEGIN
656      ASSERT( lu.ok, 20 );
657      res := lu.sign;
658      FOR n := 0 TO lu.dim - 1 DO
659          res := res * lu.amax[ n ] * lu.U[ n, n ]
660      END;
661      RETURN res
662  END Det;
663
664  PROCEDURE ( lu: LU ) Solve ( IN y: Vector; OUT x: Vector );
665      VAR dim, i, ip, j, jp, k: INTEGER; s: REAL; z: POINTER TO Vector;
666  BEGIN
667      ASSERT( lu.ok, 20 );
668      dim := lu.dim; ASSERT( LEN( y ) >= dim, 21 ); ASSERT( LEN( x ) >= dim, 22 );
669      z := lu.tmp;
670      (* первый шаг — формула (L1) в документации: *)
671      FOR i := 0 TO dim - 1 DO
672          ip := lu.pi[ i ];
673          s := y[ ip ] / lu.amax[ ip ]; (* ybar[ i ] *)
674          FOR k := 0 TO i - 1 DO
675              s := s - lu.L[ i, k ] * z[ k ]
676          END;
677          z[ i ] := s
678      END;
679      (* второй шаг — формула (U1) в документации: *)
680      FOR k := dim - 1 TO 0 BY -1 DO
681          s := z[ k ];
682          FOR j := k + 1 TO dim - 1 DO
683              s := s - lu.U[ k, j ] * x[ lu.pj[ j ] ] (* xbar[ j ] *)
684          END;
685          x[ lu.pj[ k ] ] := s / lu.U[ k, k ]
686      END;
687  END Solve;
688
689  PROCEDURE ( lu: LU ) Invert ( OUT inv: Matrix );
690      VAR dim, i, j: INTEGER; z: POINTER TO Vector;
691  BEGIN
692      ASSERT( lu.ok, 20 );
693      dim := lu.dim;
694      ASSERT( ( dim <= LEN( inv, 0 ) ) & ( dim <= LEN( inv, 1 ) ), 30 );
695      NEW( z, dim );
696      FOR j := 0 TO dim - 1 DO
697          FOR i := 0 TO dim - 1 DO z[ i ] := 0 END;
698          z[ j ] := 1;
699          lu.Solve( z, z );
700          FOR i := 0 TO dim - 1 DO
701              inv[ i, j ] := z[ i ]
702          END
703      END
704  END Invert;

```

```

705  PROCEDURE DistFromUnit* ( IN c: Matrix ): REAL;
706      VAR dim, i, j: INTEGER; a, res: REAL;
707  BEGIN
708      dim := LEN( c, 0 ); ASSERT( dim = LEN( c, 1 ), 20 );
709      res := 0;
710      FOR i := 0 TO dim - 1 DO
711          FOR j := 0 TO dim - 1 DO
712              a := c[ i, j ]; IF i = j THEN a := a - 1 END;
713              res := res + a * a
714          END
715      END;
716      RETURN Math.Sqrt( res )
717  END DistFromUnit;
718
719  PROCEDURE SetDir* ( d: Directory );
720  BEGIN ASSERT( d # NIL, 20 ); dir := d;
721  END SetDir;
722
723  PROCEDURE Init;
724      VAR d: StdDirectory;
725  BEGIN
726      NEW( d ); SetDir( d )
727  END Init;
728
729  BEGIN Init
730  END NumathMatrices.
731

```