Algorithm and Software for Integration over a Convex Polyhedron

Mark Korenblit[1] and Efraim Shmerling[2]

[1] Department of Computer Science
Holon Institute of Technology
52 Golomb Str., P.O. Box 305
Holon 58102, Israel
korenblit@hit.ac.il
[2] Department of Computer Science and Mathematics
The College of Judea and Samaria
Ariel 44837, Israel
efraimsh@yahoo.com

Abstract. We present a new efficient algorithm for numerical integration over a convex polyhedron in multi-dimensional Euclidian space defined by a system of linear inequalities. The software routines which implement this algorithm are described. A numerical example of calculating an integral using these routines is given.

1 Introduction and Description of the Algorithm

This article describes a new step in the development of algorithms and software for multiple integration.

Available standard numerical routines for multiple integration enable one to integrate over cubes of the form $\left[a_1^{(0)}; a_1^{(1)} \right] \times \left[a_2^{(0)}; a_2^{(1)} \right] \times ... \times \left[a_{n-1}^{(0)}; a_{n-1}^{(1)} \right] \times \left[a_n^{(0)}; a_n^{(1)} \right]$ in the n-dimensional Euclidean space E_n, where n is small, as well as over the sets in E_n defined by $\left[f_1^{(0)}; f_1^{(1)} \right] \times \left[f_2^{(0)}; f_2^{(1)} \right] \times ... \times [f_{n-1}^{(0)};\ f_{n-1}^{(1)}] \times \left[a_n^{(0)}; a_n^{(1)} \right]$, where $f_k^{(0)}$, $f_k^{(1)}$ $\left(k = \overline{1, n-1}\right)$ are functions, the only possible arguments of which are variables $x_{k-1}, x_{k-2}, ..., x_1$.

If the area of integration is more complicated, there are no available algorithms and software. There are only two universal methods that enable one to integrate over an arbitrary convex set (CS) in E_n, the boundary of which is defined by the equation $F(x_1, x_2, ..., x_n) = 0$. The first is the Monte-Carlo method [4]. The second is based on integration over a minimum n-dimensional cube C such that $CS \subset C$, and instead of the integral $\int\limits_{cs} f(x_1, x_2, ..., x_n) dx_1 ... dx_n$ the equivalent integral $\int\limits_{c} f_1(x_1, x_2, ..., x_n) dx_1 ... dx_n$ is calculated, where

$$f_1(x_1, x_2, ..., x_n) = \begin{cases} f(x_1, x_2, ..., x_n), & \text{if } (x_1, x_2, ..., x_n) \in CS \\ 0, & \text{otherwise} \end{cases}.$$

A. Iglesias and N. Takayama (Eds.): ICMS 2006, LNCS 4151, pp. 273–283, 2006.

Obviously, both methods are inefficient from the perspective of computation, and the second method requires a great number of "check on condition" operations which take a relatively long time. In the case where the domain of integration is a convex polyhedron much more efficient numerical methods can be developed.

Assuming that a convex polyhedron (CP) in E_n is defined by the system of linear inequalities over the variables $x_1, x_2, ..., x_n$, integration over the CP can be reduced to integration over a set of non-intersecting domains of the form $\left[f_1^{(0)}; f_1^{(1)} \right] \times \left[f_2^{(0)}; f_2^{(1)} \right] \times ... \times [f_{n-1}^{(0)};\ f_{n-1}^{(1)}] \times \left[a_n^{(0)}; a_n^{(1)} \right]$, where the functions $f_k^{(0)}, f_k^{(1)}$ $\left(k = \overline{1, n-1}\right)$ are linear combinations of variables $x_{k-2}, x_{k-1}, ..., x_1$.

Integration over such domains can be carried out utilizing available multiple integration routines (such routines are considered in [5] and [6]).

Integration over a convex polyhedron (and in the simplest case, calculating the volume of a convex polyhedron) has many applications. For example, it is required in solving certain probabilistic problems. Let's formulate one of them. For an n-th order random polynomial

$$a_0(\omega) + a_1(\omega)x + a_2(\omega)x^2 + ... + a_n(\omega)x^n = 0,$$

where $a_0(\omega), ..., a_n(\omega)$ are independent random variables uniformly distributed in the intervals $\left[a_1^{(0)}; a_1^{(1)} \right], \left[a_2^{(0)}; a_2^{(1)} \right], ..., \left[a_n^{(0)}; a_n^{(1)} \right]$, respectively, find the distribution of the number of real zeros of the polynomial belonging to a certain real interval $[a; b]$ (see [2]).

This problem can be easily reduced to the calculation of the volumes of a finite number of convex polyhedrons, defined by systems of linear inequalities.

An example of solving a problem of this kind with the help of the presented algorithm for multiple numerical integration is given in section 3 of the article.

In order to describe the presented method we need the following notations.

Notation 1. *Double inequality.*
Double inequality (DI) of j-th order is an ordered pair of j-th order vectors $LP = (LP_0, ..., LP_{j-1})$, $RP = (RP_0, ..., RP_{j-1})$ *which correspond to the inequality*

$$LP_0 + LP_1 \times x_1 + ... + LP_{j-1} \times x_{j-1} < x_j < RP_0 + RP_1 \times x_1 + ... + RP_{j-1} \times x_{j-1}.$$

Remark 1. From here on all the inequalities mentioned in the text are double inequalities. Assume that possible values of the variables x_i $\left(i = \overline{1, n}\right)$ are restricted to bounded intervals $[LConst_i; RConst_i]$ $\left(i = \overline{1, n}\right)$, respectively, this enables one to rewrite each linear inequality as a DI which is a necessary preliminary step for the algorithm.

Notation 2. *System of r-type.*
We say the system of inequalities belongs to r-type $(2 \le r \le n)$ *if it contains exactly one inequality of order i for each i such that* $r < i \le n$ *and more than one r-th order inequalities. The system of inequalities belongs to a one-type if it has exactly one inequality of order i for each i such that* $1 \le i \le n$.

Notation 3. *Transformation of an ordered pair* (LV, RV) *of vectors of* i*-th order into DI of order* k $(k \le i-1)$.

The inequality corresponding to the pair (LV, RV) *is of the form*

$$LV_0 + LV_1 \times x_1 + ... + LV_{i-1} \times x_{i-1} < RV_0 + RV_1 \times x_1 + ... + RV_{i-1} \times x_{i-1} \quad (1)$$

If $LV_{i-1} < RV_{i-1}$*, we rewrite (1) in the form*

$$RConst_{i-1} > x_{i-1} > \frac{LV_0 - RV_0}{RV_{i-1} - LV_{i-1}} + \frac{LV_1 - RV_1}{RV_{i-1} - LV_{i-1}} \times x_1 + \\ + \frac{LV_2 - RV_2}{RV_{i-1} - LV_{i-1}} \times x_2 + ... \frac{LV_{i-2} - RV_{i-2}}{RV_{i-1} - LV_{i-1}} \times x_{i-2}. \quad (2)$$

DI corresponding to (2) is then the result of the transformation.
If $LV_{i-1} > RV_{i-1}$*, we rewrite (1) in the form*

$$LConst < x_{i-1} < \frac{RV_0 - LV_0}{RV_{i-1} - LV_{i-1}} + \frac{RV_1 - LV_1}{RV_{i-1} - LV_{i-1}} \times x_1 + \\ + \frac{RV_2 - LV_2}{RV_{i-1} - LV_{i-1}} \times x_2 + ... + \frac{RVi_{-2} - LV_{i-2}}{RV_{i-1} - LV_{i-1}} \times x_{i-2}. \quad (3)$$

If $LV_{i-k} = RV_{i-k}$, $k = 1, ..., k_1$, $LV_{i-k_1-1} \ne RV_{i-k_1-1}$,*we transform the pair of vectors* $(LV_0, LV_1, ..., LV_{i-k_1-1})$ *of order* $i - k_1$ *into DI of order* $i - k_1 - 1$ *via formulae (2) or (3), where instead of* i *we substitute* $i - k_1$.

Notation 4. *Decomposition.*
Assume that the initial system of inequalities (IS) belongs to r*-type. The system has at least two inequalities of* r*-th order. These inequalities have ndlp distinct left parts and ndrp distinct right parts. So we have a pair of vectors* (LP, RP)*, which consist of distinct left parts and distinct right parts, respectively, of* r*-th order inequalities belonging to the system.*

The IS is decomposed into $ndlp \times ndrp$ *systems* S_{ij}, $i = \overline{1, ndlp}$, $j = \overline{1, ndrp}$. *Each of the systems* S_{ij} *is formed in the following way:*

(a) All inequalities (LP_i, RP_j) *of orders different from* r *which belong to IS are included in* S_{ij}*;*

(b) Only one inequality (LP_i, RP_j) *of order* r *is included in* S_{ij}*;*

(c) All the inequalities obtained via transformations of all pairs of vectors
(LP_i, RP_j), $s \ne i$, $s = \overline{1, ndlp}$
(RP_j, RP_p), $p \ne j$, $p = \overline{1, ndrp}$
are included in S_{ij}*;*

(d) The inequality obtained via the transformation of the pair of vectors (LP_i, RP_j) *is included in* S_{ij}.

The systems S_{ij} *which have no solutions (we call them invalid) are ignored.*

It can be easily shown that:

- The system $\{S_{ij}\}_{i=\overline{ndlp}}$ is equivalent to IS.
- The system $\{S_{ij}\}_{i=\overline{ndlp}}^{j=\overline{ndrp}}$ consists of nonintersecting systems of inequalities.
- The types of systems S_{ij} are less than or equal to $r-1$.

Algorithm 1. *Algorithm for Integration over a Convex Polyhedron*

Step 1. Set $Sum = 0$ *(initialization of the calculated integral).*

Step 2. Check what is the type of IS. If the IS belongs to a one-type, go to step 3, otherwise go to step 4.

Step 3. Integrate over the set defined by the IS, add the calculated integral to Sum and terminate.

Step 4. Decompose the IS into nonintersecting systems and for each of them go to step 2 viewing it as IS.

Any available routine for numerical integration can be applied in step 3 of this algorithm.

2 Software Routines for the Algorithm Realization

The algorithm has been programmed in C++ language.

We define three basic structures named **Inequality**, **SOKOI** (system of k-th order inequalities), and **SOI** (system of inequalities). Their description is presented in Listing 1.

Structure **Inequality** implements an inequality of order k. Structure **SOKOI** realizes a system of m inequalities, each one of order k. Structure **SOI** includes n **SOKOI** structures, each of which presents a group of m_j inequalities of order j, $j = 1, 2, \ldots, n$. Thus, **SOI** forms a general system of inequalities.

Listing 1. `Basic Structures`

```
/* Implementation of inequality of order k */
struct Inequality
{
    int coefficients_num; // order of inequality
                          // (number of coefficients)
    double *left, *right; // pointers to left and right vectors
    Inequality (int k=0) // constructor
        : left (NULL),
          right (NULL)
    {
        coefficients_num = k; // set number of coefficients
```

```
        if (coefficients_num > 0)
        {
            // dynamic allocation of a left vector
            left = new double[coefficients_num];
            // dynamic allocation of a right vector
            right = new double[coefficients_num];
        }
    } // constructor Inequality
}; // struct Inequality

/* Implementation of a system of m inequalities,
   each one of order k */
struct SOKOI // SOKOI - system of k-th order inequalities
{
    int inequalities_num, // number of inequalities in a system
        coefficients_num; // order of inequality
                          // (number of coefficients in inequality)
    Inequality *inequalities_vector; // pointer to array
                                     // of inequalities
    SOKOI (int m=0, int k=0) // constructor
        : inequalities_vector (NULL)
    {
        inequalities_num = m; // set number of inequalities
        coefficients_num = k; // set number of coefficients
                              // in inequality
        if (inequalities_num > 0)
        {    // dynamic allocation of array of inequalities
            inequalities_vector = new Inequality[inequalities_num];
            // initialization of array of inequalities
            for (int i=0; i < inequalities_num; i++)
                inequalities_vector[i] =
                                  Inequality (coefficients_num);
        }
    } // constructor SOKOI
}; // struct SOKOI

/* Implementation of n-th order system of inequalities */
struct SOI // SOI - system of inequalities
{
    int levels_num, // order of system (number of levels)
        *inequalities_num_in_level; // pointer to array of
                                    // numbers of inequalities
                                    // in each level
    SOKOI *systems_vector; // pointer to array of SOKOIs;
            // each SOKOI implements a subsystem consisting
            // of inequalities of equal orders
            // (inequalities which are in the same level)
    SOI (int n=0, int m[] = NULL) // constructor
        : systems_vector (NULL)
    {
        levels_num = n; // set number of levels
```

```
        inequalities_num_in_level = m; // set pointer to array of
                                       // numbers of inequalities
                                       // in each level
        if (levels_num > 0)
        {    // dynamic allocation of array of SOKOIs
            systems_vector = new SOKOI[levels_num + 1];
            // initialization of array of SOKOIs
            for (int j=1; j <= levels_num; j++)
                systems_vector[j] =
                        SOKOI (inequalities_num_in_level[j], j);
        }
    } // constructor SOI
}; // struct SOI
```

The following basic functions are used:

- double SKoAl (SOI system)
- double Recursion (SOI *father_system, int level)
- bool VectorsEqual (double vect1[], double vect2[], int size)
- bool Transform (int level, int left_num, int right_num, SOI old_system, SOI &new_system)
- void DeleteSoi (SOI *ptr_SOI)
- double Integration (SOI system)

In addition, the following auxiliary functions are utilized:

- void Auxiliary (double in_left[], double in_right[], int in_level, Inequality &new_inequality, int &out_level)
- void CopyIneq (Inequality in_inequality, Inequality &out_inequality)
- void DeleteIneq (Inequality inequality)
- bool FirstLevel (double old_left_0, double old_right_0, double old_left_1, double old_right_1, SOI &new_system)

SKoAl is the basic function which realizes the algorithm. Its input is a system of inequalities (type **SOI**) and its output is a numeric value of the integral.

As follows from Algorithm 1, the algorithm is an iterative process accompanied by the decreasing of parameter r (r determines the type of the system). This iterative process is provided by function **Recursion** (see Listing 2) which is a basis for function **SKoAl**.

Listing 2. `double Recursion (SOI *father_system, int level)`

```
{
    double sum = 0; // initialization of the calculated integral
    if (level > 1) // system is not one-type
    {
        /* Scanning all left and right parts */
        for (int j = 0; // scanning right parts
                 j < (*father_system).inequalities_num_in_level
                                                    [level];
                 j++)
        {
```

```
// scanning all previous right parts
for (int kj = 0; kj < j; kj++)
    // comparison of a right part with previous one
    if (VectorsEqual ( (*father_system).
                            systems_vector[level].
                              inequalities_vector[kj].
                                right,
                           (*father_system).
                              systems_vector[level].
                                inequalities_vector[j].
                                  right,
                           level ) )
        break; // from "for kj"
// loop "for kj" wasn't completed because of
// finding identical right parts
if ( kj < j)
    continue; // "for j"
for (int i = 0; // scanning left parts
     i < (*father_system).inequalities_num_in_level
                                              [level];
          i++)
{
    // scanning all previous left parts
    for (int ki = 0; ki < i; ki++)
    // comparison of a left part with previous one
        if (VectorsEqual ( (*father_system).
                            systems_vector[level].
                              inequalities_vector[ki].
                                    left,
                              (*father_system).
                             systems_vector[level].
                               inequalities_vector[i].
                                     left,
                              level ) )
            break; // from "for ki"
    // loop "for ki" wasn't completed because of
    // finding identical left parts
    if (ki < i)
        continue; // "for i"
SOI *child_system = new SOI; // memory allocation
                             // for a new system
    // generating a new system that is
    // nearer to one-type
    if ( Transform (level, i, j,*father_system,
                                 *child_system) )
        // recursive call on new system and adding
        // calculated integral to sum
        sum += Recursion (child_system, level - 1);
    DeleteSoi (child_system); // free a space
                              // of the new system
```

```
            } // "for i"
        } // "for j"
        return sum;
    } // if (level > 1)
    else // level == 1 (system is one-type)
        // integration over set defined by one-type system
        return Integration (*father_system);
}
```

Function **Recursion** constructs a tree in such a way that each node of this tree is an IS. The root of the tree is an original system of inequalities which enters function **SKoAl**. The tree leaves are systems of inequalities which belong to a one-type. Input parameters of **Recursion** are a pointer to **SOI** named **father_system** and an integer variable **level** that is equal to r.

In order to avoid generating duplicates of systems due to identical vectors, we use a boolean function, **VectorsEqual** in the course of the execution of **Recursion**. Input parameters of **VectorsEqual** are two numeric vectors and their size. This function compares the left/right part of an inequality currently under consideration with the left/right parts (respectively) of previously considered inequalities of the same system (as noted in Notation 4, we consider only distinct left parts and only distinct right parts.). If the corresponding parts are identical, then the next left/right part is compared with previously considered ones. If the corresponding parts are not identical, then the memory for a child of a node which is currently under consideration is allocated dynamically by means of a pointer to **SOI** named **child_system**.

The child node is intended for one of the new nonintersecting systems generated in step 4 of Algorithm 1. This system is constructed by means of a boolean function, **Transform**. The parameters of this function are (a) **level**; (b) integer variables **left_num** and **right_num** (which are substituted by the current numbers of the left and right parts, respectively, of the r-th order inequalities (see Notation 4)); and (c) two systems of inequalities (type **SOI**) named **old_system** and **new_system**. Function **Transform** is invoked by substituting structures pointed by pointers **father_system** and **child_system** instead of parameters **old_system** and **new_system**, respectively.

If a new system which is constructed by **Transform** is invalid, then the function returns $false$. Otherwise, **Transform** returns $true$, and the next recursion step of the algorithm is performed by the call of function **Recursion** with parameters **child_system** and **level** $-$ **1**. After return from this recursive call, the returned value is added to the local variable, **sum** (according to step 3 of Algorithm 1). Then (or immediately after the call of **Transform** if it returns $false$) function **DeleteSoi** frees a space allocated for a child of a currently considered node, and the value accumulated in **sum** is returned.

Recursive calls terminate when the value of **level** is equal to one, i.e., for the system belonging to a one-type. In this case, step 3 of Algorithm 1 is performed. Function **Integration** integrates over the set which is defined by the corresponding system and returns the calculated value to **Recursion**. Then, **Recursion** returns this value to its previous copy, into which this value is added to **sum**.

The final value of **sum** is returned by **Recursion** to function **SKoAl**, and is returned by **SKoAl** as a result.

Other functions are used in function **Transform**.

Function **Auxiliary** realizes transformation of an ordered pair of vectors into the double inequality (this procedure is described in Notation 3). Its input parameters named **in_left**, **in_right**, and **in_level**, correspond to the pair of vectors and to its order, respectively. The output parameters are structure **new_inequality** of type **Inequality** and integer variable **out_level**. They correspond to the constructed double inequality and to its order, respectively.

Function **CopyIneq** copies content of an input structure **in_inequality** (type **Inequality**) into a new structure **out_inequality**. This function is used for copying inequalities into **new_system** of function **Transform**.

Function **DeleteIneq** frees a space allocated for temporary objects of type **Inequality** in function **Transform**. This function is applied in function **DeleteSoi** as well.

A boolean function, **FirstLevel** constructs an inequality of the first order in a system which is generated by function **Transform**. If this inequality has no solutions, then **FirstLevel** returns $false$ into **Transform**, and the corresponding system is declared invalid.

The iterative process provided by function **Recursion** can be visualized as a recursion tree the nodes of which are recursive calls. And in this particular case, each recursion step includes a dynamic allocation. That is, as stated above, function **Recursion** also constructs a real tree the nodes of which are systems of inequalities implemented by structures of type **SOI**. But we need not keep all the nodes of this tree in the memory simultaneously. Before generating a new system of r-type, the previous r-type system is erased from memory. Thus, at most n nodes (a path from the the root to a leaf) are kept in memory simultaneously.

In fact, our algorithm is based on depth-first search (DFS) (see [1], [3]). However, in this particular case, a node is created directly before it is visited. The call of **Transform** corresponds to the visit. After all children of a node have been visited, the node is deleted by **DeleteSoi**.

Let's estimate the maximum number of nodes in our tree. Consider the n-th order system that has m inequalities. Suppose all m inequalities are of order n, i.e., the root of the tree has the maximum size which is possible. Also, suppose that a system of r-type generates only systems of $r-1$-type in every recursion step. The last supposition is that function **VectorsEqual** always returns $false$, i.e., $ndlp$ is equal to $ndrp$ all the time. In such a case, the number of nodes is

$$1+\sum_{i=2}^{n}\prod_{j=2}^{i}\left(2^{j-2}m-2^{j-2}+1\right)^2>\sum_{i=1}^{n}m^{2(i-1)}=\frac{m^{2n}-1}{m^2-1}.$$

However, the actual number of nodes which are kept in memory simultaneously is determined by the height of the tree and does not exceed n. It is clear that the sizes of these nodes are different. Corresponding computations show that in this case, the amount of required memory is $O\left(m2^n\right)$.

3 The Illustrative Example

In order to utilize the software described in section 2 for calculating volumes of convex polyhedrons in E_4, the simplified version of function **Integration** (mentioned in the previous section) which calculates the integral

$$\int_{a_1}^{a_2} \int_{a_3+a_4x_1}^{a_5+a_6x_1} \int_{a_7+a_8x_1+a_9x_2}^{a_{10}+a_{11}x_1+a_{12}x_2} \int_{a_{13}+a_{14}x_1+a_{15}x_2+a_{16}x_3}^{a_{17}+a_{18}x_1+a_{19}x_2+a_{20}x_3} 1 \; dx_4 dx_3 dx_2 dx_1$$

can be applied. This integral is the volume of the convex set corresponding to the following one-type system of inequalities:

$$\begin{cases} ([a_{13}, a_{14}, a_{15}, a_{16}], [a_{17}, a_{18}, a_{19}, a_{20}]) \\ ([a_7, a_8, a_9], [a_{10}, a_{11}, a_{12}]) \\ ([a_3, a_4], [a_5, a_6]) \\ ([a_1], [a_2]) \end{cases}.$$

With the help of these tools, we solve the following simple probabilistic problem.

Problem 1. Calculate the probability that the polynomial equation with random coefficients

$$c_1(\omega) + c_2(\omega)\, x + c_3(\omega)\, x^2 + c_4(\omega)\, x^3 = 0,$$

where $c_1(\omega)$, $c_2(\omega)$, $c_3(\omega)$, $c_4(\omega)$ are independent random variables uniformly distributed in the interval $[-1; 1]$, has one or three real roots belonging to the interval $[0; 1]$.

Obviously, the problem can be reduced to calculating the volumes of two convex sets in E_4 (the volumes are equal). The first set is defined by the system of inequalities

$$\begin{cases} c_1 + c_2 + c_3 + c_4 > 0 \\ c_1 < 0 \\ -1 < c_1 < 1 \\ -1 < c_2 < 1 \\ -1 < c_3 < 1 \\ -1 < c_4 < 1 \end{cases}.$$

The second one is

$$\begin{cases} c_1 + c_2 + c_3 + c_4 < 0 \\ c_1 > 0 \\ -1 < c_1 < 1 \\ -1 < c_2 < 1 \\ -1 < c_3 < 1 \\ -1 < c_4 < 1 \end{cases}.$$

The first system can be rewritten in the following form:

$$\begin{cases} -1 < c_4 < 1 \\ -c_1 - c_2 - c_3 < c_4 < 1 \\ -1 < c_3 < 1 \\ -1 < c_2 < 1 \\ -1 < c_1 < 0 \end{cases} .$$

This system has been expressed as a system of double inequalities. The volume calculated by means of our software is equal to 2.79167. This result has been checked by the Monte-Carlo method and appears to be correct.

4 Conclusion and Future Work

We have presented a new algorithm for numerical integration over a convex polyhedron in the n-dimensional Euclidean space defined by a system of linear inequalities. We have described the software routines implementing the algorithm and estimated the memory costs required for their realization. We intend to estimate the running time of the algorithm as well. We also intend to determine maximum values of both n and the number of inequalities for which our algorithm is practically realizable. In addition, we plan to compare our algorithm with alternative methods based on various criteria.

References

1. *Algorithms and Theory of Computation Handbook*, edited by M. J. Atallah. CRC Press, Boca Raton (1999)
2. A. T. Bharucha-Reid and M. Sambandham: *Random Polynomials*. Academic Press (1986)
3. T. H. Cormen, C. E. Leiseron, and R. L. Rivest: *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts (1994)
4. M. H. Kalos and P. A. Whitlock: *Monte Carlo Methods*. John Wiley & Sons (1986)
5. W. H. Press, S. A. Tenkovsky, W. T. Vetterling, and B. P. Flannery: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press (1992)
6. J. R. Price: *Numerical Methods, Software and Analysis*. Academic Press (1993)