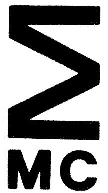
stichting
mathematisch
centrum



AFDELING NUMERIEKE WISKUNDE NW 99/80 DECEMBER
(DEPARTMENT OF NUMERICAL MATHEMATICS)

H. SCHIPPERS

THE AUTOMATIC SOLUTION OF FREDHOLM EQUATIONS OF THE SECOND KIND

Preprint

kruislaan 413 1098 SJ amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 65X04, 65F10, 65R20

The automatic solution of Fredholm equations of the second kind *)

by

H. Schippers

ABSTRACT

For the automatic solution of Fredholm equations of the second kind a new code, called *solve int eq,* is presented. The linear system resulting from the discretization of the integral equation is iteratively solved by a multiple grid method. We selected this method because in a previous paper [4] we have shown that multiple grid processes take only $\mathcal{O}(N^2)$ arithmetic operations. For a variety of problems the performance of *solve int eq* is compared with Atkinson's program *iesimp* [1]. This program appears to be about 50% more expensive than *solve int eq*. Additionally, *solve int eq* is applied to the aerodynamic problem of calculation of potential flow around a Kármán-Trefftz aerofoil.

KEY WORDS & PHRASES: *Fredholm integral equations of the second kind, Multiple grid methods, automatic algorithm*

---

*) This report will be submitted for publication elsewhere.

1. INTRODUCTION

In the present paper we describe an algorithm for the automatic solution of Fredholm equations of the second kind:

$$(1.1) \qquad f(x) - \int_a^b K(x,y)f(y)\,dy = g(x), \qquad x \in [a,b].$$

The algorithm is an improvement of Atkinson's automatic program *iesimp* [1] in the sense that a new iterative method is used for the solution of the non-sparse systems of equations that arise from the approximation of (1.1). Our iterative methods are multiple grid methods that work with a sequence of grids of increasing refinement. These grids are simultaneously used to obtain an approximation to the original continuous problem (1.1). The multiple grid methods used can be seen as an extension of Atkinson's iterative scheme, that uses only two grids: a coarse and a finer grid. Convergence and computational complexity of multiple grid methods have been studied in a previous paper [4]. The program has been written in the algorithmic language ALGOL 68, since in this language we can easily and efficiently handle the data structures and the recursive procedures that appear in multiple grid methods.

A description of our multiple grid methods can be given by collectively compact operators and interpolatory projections onto subspaces of piecewise continuous functions. This has been done in section 2, where also some results from [4] are collected. Based on the theoretical foundation of section 2, the program for the automatic solution of Fredholm equations, *solve int eq*, is described in section 3. Numerical examples illustrating the method are given in section 4. Comparisons have been made with Atkinson's automatic program *iesimp*.

2. DESCRIPTION OF MULTIPLE GRID METHODS

In this section we write equation (1.1) in operator notation as

$$(2.1) \qquad (I-K)f = g, \qquad g \in X,$$

where X is a Banach space and $K: X \to X$ the linear operator associated with

the kernel $k(x,y)$. It is assumed that 1 is not an eigenvalue of K. Thus, therefore (I-K) has a bounded inverse on X. We approximate the solution of (2.1) by a sequence of interpolating spline functions f_p with knots at the points $G_p = \{t_i | a = t_0 < t_1 \ldots < t_{N_p} = b\}$. On the interval [a,b] all grids $\{G_p\}, p = 0,1,2,\ldots \ell$ are selected such that $G_0 \subset G_1 \subset \ldots \subset G_\ell$. In the context of multiple grid iteration, the subscript p is called "level"; h_p is a measure of the mesh-size defined by:

$$h_p = \max_{t_i \in G_p} |t_i - t_{i-1}|.$$

In our algorithm we take the sequence of grids $\{G_p\}$ uniform with $N_p = 2^p N_0$, so that

B1. $$h_p = 2^{-p} h_0.$$

Corresponding with the sequence $\{h_p\}$ we approximate K by a sequence of approximating operators $\{K_p\}$, $K_p: X \to X$.
Let X_p, $p = 0,1,2,\ldots$, be the finite-dimensional subspaces of interpolating spline functions and let T_p, $p = 0,1,2,\ldots$, be the interpolating operators. We use the following assumptions on $\{K_p\}$ and $\{T_p\}$:

B2. $$K_p = K_p T_p,$$

B3. $$\|(K-K_p)K\| \le C_1 h_p^\alpha, \quad \alpha > 0,$$

B4. $$\|(I-T_p)K\| \le C_2 h_p^\beta, \quad \beta > 0.$$

Although the above assumptions are not necessary conditions, they are sufficient to satisfy the assumptions A1-A6 of section 2 in our previous paper [4]. Hence, the theory presented in [4] applies to our present algorithm.

Let X be the Banach space C[a,b], provided with its supremum norm, and let $y \in C[a,b]$. If Ky is twice (respectively four times) continuously differentiable the assumptions B2-B4 can be verified for the following examples 1 and 2.

EXAMPLE 1. The operator K_p is defined by the repeated trapezoidal rule and

the operator T_p by continuous, piecewise linear interpolation, in which case $\alpha = \beta = 2$.

EXAMPLE 2. K_p is defined by the repeated Simpson's rule and T_p by continuous, piecewise cubic interpolation. In this case $\alpha = \beta = 4$.

EXAMPLE 3. *Finite element methods for integral equations from potential theory*. Let D be a simply connected finite plane region bounded by a smooth contour S with continuous curvature. S is given by the parametric equations $x = X(s)$, $y = Y(s)$, $s \in [0,1]$. The kernel function is given by

$$k(s,t) = -\frac{1}{\pi}\frac{d}{dt}\arctan\left(\frac{Y(t)-Y(s)}{X(t)-X(s)}\right).$$

Define the operator T_p by piecewise constant interpolation at the mid-points $t_{i+\frac{1}{2}} = (t_i+t_{i+1})/2$, $i = 0,\ldots,N_p-1$. Let K_p be defined by KT_p. The space X must be chosen such that for each N it contains the class of piecewise constant functions. Following SLOAN [10] we choose X to be the Banach space $Z[0,1]$, which is the closure (in the supremum norm) of the space of piecewise continuous functions on [0,1] which satisfy

$$f(t) = \tfrac{1}{2}[\lim_{s\to t^-} f(s) + \lim_{s\leftarrow t^+} f(s)], \qquad t \in (0,1),$$

$$f(0) = \lim_{s\to 0^+} f(s), \qquad f(1) = \lim_{s\to 1^-} f(s).$$

In this case we get $\alpha = 1 + \rho$, where ρ is a measure for the smoothness of S $(0<\rho\leq 1)$ and $\beta = 1$. See [8].

On level p we wish to approximate the solution of equation (2.1) by:

$$A_p f_p = T_p g, \qquad f_p \in X_p, \tag{2.2}$$

where $A_p = I-T_pK_p$. We assume that the mesh-size h_0 is sufficiently small such that A_p^{-1} exists for all $p \geq 0$. If the forcing-function $g(x)$ is several times continuously differentiable such that $\|(K-K_p)g\| \leq C_3h_p^\alpha$, it follows from assumption B3 that $\|(K-K_p)f\| \leq C_4h_p^\alpha$. From the work of PRENTER [6] the following error estimate is obtained:

$$(2.3) \qquad \|T_p f - f_p\| \le C_5 \|(K-K_p)f\| \le C_4 C_5 h_p^{\alpha}.$$

In *solve int eq* we use this asymptotic behaviour of the error to extrapolate and predict the size of the error for small values of h_p.

The solution $f_p \in X_p$ of (2.2) is approximated by a defect correction process of the form

$$(2.4) \qquad \begin{cases} f_{p,0} = 0, \\ f_{p,i+1} = B_p^{(j)} g_p + (I - B_p^{(j)} A_p) f_{p,i}, \end{cases}$$

where $B_p^{(j)}$, $(j = 1,2,3)$, is an approximate inverse of A_p and $g_p = T_p g$. In [4, section 4] we studied the convergence properties of (2.4) with respect to the following choices for $B_p^{(j)}$:

$$B_p^{(1)} = T_p + T_{p-1} A_{p-1}^{-1} T_p K_p,$$

$$\begin{cases} B_0^{(2)} = T_0 A_0^{-1}, \\ B_p^{(2)} = T_p + T_{p-1} Q_{p-1}^{(2)} T_p K_p, \end{cases}$$

$$\begin{cases} B_0^{(3)} = T_0 A_0^{-1} \\ B_p^{(3)} = T_p - T_{p-1} + T_{p-1} Q_{p-1}^{(3)} (T_{p-1} - T_{p-1} K_{p-1} + T_p K_p), \end{cases}$$

with

$$Q_p^{(j)} = \sum_{m=0}^{\gamma-1} (T_p - B_p^{(j)} A_p)^m \, B_p^{(j)}, \qquad j = 2,3,$$

for some positive integer γ.

We notice that $B_p^{(1)}$ is only of theoretical value, since the dimension of A_{p-1}^{-1} tends to infinity as $p \to \infty$. The operator $B_p^{(2)}$ yields an iterative process, that is equivalent with the multi grid method discussed in HACKBUSCH [3], whereas $B_p^{(3)}$ yields a multiple grid method with better convergence properties for integral operators with a large value of $\|K\|$.

For the various iterative processes we denote the rate of convergence of (2.4) by $\eta_p^{(j)}$. Since the convergence of (2.4) depends on the Lipschitz

constant of the operator $I - B_p^{(j)}A_p$ as a mapping from X_p onto X_p, it follows that

$$\eta_p^{(j)} = \|T_p(I-B_p^{(j)}A_p)\|.$$

THEOREM 2.1.

i. $\eta_p^{(1)} \leq C_3 h_p^{\min(\alpha,\beta)}$,

ii. $\eta_p^{(2)} \leq \eta_p^{(1)} + \eta_{p-1}^{(2)\gamma}[\eta_p^{(1)} + \|T_p\|\|K_p\|]$,

iii. $\eta_p^{(3)} \leq \eta_p^{(1)} + \eta_{p-1}^{(3)\gamma}[\eta_p^{(1)} + \|T_p\|]$.

PROOF. See [4, theorems 4.1 and 4.2]. □

Based on part (i) of this theorem, the defect correction process (2.4) induced by $B_p^{(1)}$ can be expected to have a regular geometric rate of convergence, i.e.

$$\|f_{p,i+1}-f_p\|/\|f_{p,i}-f_p\| \to v_p \quad \text{as } i \to \infty,$$

where the constant $v_p \to 0$ as $p \to \infty$. By assumption B1 it follows that $v_p = v_1 d^{p-1}$ with $d = 2^{-\min(\alpha,\beta)}$.

The multiple grid processes are constructed in such a way that $B_1^{(1)} = B_1^{(2)} = B_1^{(3)}$. As a consequence the rates of convergence of the multiple grid processes depend on the magnitude of v_1, γ, $\|T_p\|$ and $\|K_p\|$. For $j = 2,3$, this dependence is explained by means of the definition of the sequence $\{w_p^{(j)}\}$ given by:

$$(2.5) \qquad \begin{cases} w_1^{(j)} = v_1, & v_p = d^{p-1}v_1, \\ w_p^{(j)} = v_p + w_{p-1}^{(j)\gamma}(v_p+k_j), & p > 1, \end{cases}$$

where

$$k_j = \begin{cases} \|T_p\|\|K_p\|, & j = 2 \\ \|T_p\| \quad , & j = 3. \end{cases}$$

If $\eta_p^{(1)} \leq v_p$, by induction it may be shown that $\eta_p^{(j)} \leq w_p^{(j)}$, $(j = 2,3)$. In the following lemma we formulate a condition on v_1, such that $w_p^2 < v_1 v_p$, i.e. two multiple grid iteration sweeps yield an iterative approximation for f_p, which has the same geometric rate of convergence as one application of (2.4) with approximate inverse $B_p^{(1)}$.

<u>LEMMA 2.2</u>. *Let* v_p *and* $w_p^{(j)}$ $(j = 2$ *or* $3)$ *be defined by* (2.5) *and let* $\gamma = 2$, $v_1 < 1$, *then* ${w_p^{(j)}}^2 \leq v_1 v_p$ *if and only if*

$$d + v_1(v_2+k_j) < \sqrt{d}. \tag{2.6}$$

<u>PROOF</u>. Since $\alpha,\beta > 0$ it follows that $d = 2^{-\min(\alpha,\beta)} < 1$, so that $v_p < v_{p-1} < \ldots < v_1 < 1$. By definition (2.5) we have $w_1^{(2)} = w_1^{(3)} = v_1$ and by induction we obtain for $p > 1$:

$$\begin{aligned} w_p^{(j)} &= v_p + {w_{p-1}^{(j)}}^2 (v_p+k_j), \\ &\leq v_p + v_1 v_{p-1}(v_2+k_j), \\ &\leq \frac{v_p}{d}(d+v_1(v_2+k_j)). \end{aligned}$$

Substituting condition (2.6) we get for $p > 1$:

$${w_p^{(j)}}^2 < \frac{1}{d} v_p^2 < v_1 v_p.$$

Conversely, we first note that $v_p \leq w_p^{(j)}$. Hence, for $p > 1$

$$v_p + v_{p-1}^2(v_p+k_j) \leq w_p^{(j)}.$$

Since ${w_p^{(j)}}^2 < v_1 v_p$ it follows that

$$\begin{aligned} &\{v_p+v_{p-1}^2(v_p+k_j)\}^2 < v_1 v_p, \\ &v_{p-1}^2\{d+v_{p-1}(v_p+k_j)\}^2 < v_1 d v_{p-1}, \\ &\{d+v_{p-1}(v_p+k_j)\}^2 < d^{3-p}. \end{aligned}$$

For increasing values of p the left-hand side decreases, whereas the right-hand side increases. However, the inequality should hold for all values of p, in particular for p = 2. In that case condition (2.6) is obtained. □

As a consequence of this lemma condition (2.6) is sufficient to show that the rates of convergence of the multiple grid processes satisfy

$$\eta_p^{(j)^2} < v_1 v_p, \qquad j = 2,3.$$

For different values of d and k_j the upper bounds on v_1 are given in table 2.1.

d \ k_j	1	6	24	100
1/16	1.85 - 1	3.12 - 2	7.81 - 3	1.87 - 3
1/4	2.36 - 1	4.16 - 2	1.04 - 2	2.50 - 3
1/2	1.89 - 1	3.44 - 2	8.63 - 3	2.07 - 3
3/4	1.07 - 1	1.93 - 2	4.83 - 3	1.16 - 3

Table 2.1 Upperbounds on v_1 to obtain $w_p^2 \le v_1 v_p$, with $\gamma = 2$.

The upperbounds on v_1 are essentially the requirement of "a fine enough mesh" in the coarsest discretization of the multiple grid algorithm. Since on the coarsest grid the system of equations in solved by a direct method (e.g. Gauss-elimination) we like to have v_1 as large as possible, so that a more robust algorithm is obtained. Therefore, in *solve int eq* the condition (2.6) on v_1 is replaced by a weaker one, which also guarantees fast convergence. We start on some coarse grid and we estimate v_1. A test is made to check whether

$$\text{(2.7)} \qquad w_p^{(j)^2} < v_p, \qquad p = 1,2,\ldots,\ell.$$

The necessary upper bound on v_1 again depends on the actual value of k_j and d as is shown in table 2.2.

$d \backslash k_j$	1	6	24	100
1/16	3.51 - 1	9.11 - 2	2.81 - 2	7.46 - 3
1/4	4.47 - 1	1.17 - 1	3.27 - 2	8.27 - 3
1/2	4.34 - 1	1.10 - 1	2.93 - 2	7.13 - 3
3/4	3.70 - 1	8.82 - 2	2.25 - 2	5.40 - 3

Table 2.2. Upper bounds on v_1 to obtain ${w_p^{(j)}}^2 \le v_p$, with $\gamma = 2$.

From table 2.2 we conclude that the condition on v_1 becomes stronger as k_j becomes larger. Since $k_2 = \|T_p\|\|K_p\|$ and $k_3 = \|T_p\|$ the rate of convergence of the multiple grid process defined by $B_p^{(3)}$ does not depend on $\|K_p\|$. Therefore, in our code we apply the approximate inverse $B_p^{(3)}$ rather than $B_p^{(2)}$. Without confusion we further use η_p and w_p for $\eta_p^{(3)}$ and $w_p^{(3)}$, respectively. Since $\eta_p \le w_p$ it follows that

$$(2.8) \qquad \|f_{p,i+1}-f_{p,i}\| \le w_p\|f_{p,i}-f_p\|,$$

i.e. the multiple grid process has a regular geometric rate of convergence. The constant w_p follows from (2.5) as soon as v_1 has been determined.

After σ iteration sweeps the multiple grid proces yields an approximate solution $f_{p,\sigma}$ for which the following error estimate holds

$$(2.9) \qquad \|T_pf-f_{p,\sigma}\| \le \|T_pf-f_p\| + \|f_{p,\sigma}-f_p\|.$$

In our code *solve integ* we determine the integers p and σ in an automatic way so that $\|T_pf-f_{p,\sigma}\|$ is less than a prescribed value *tol*. This is achieved by estimating the errors on the right-hand side of (2.9). Asymptotically for $p \to \infty$, condition (2.7) insures that only two multiple grid sweeps yield a result of the order of the approximation error $\|T_pf-f_p\|$. On the lower levels (i.e. small p) we determine σ so that

$$(2.10) \qquad \|f_{p,\sigma}-f_p\| \le 0.1*\|T_pf-f_p\|,$$

i.e. the iteration error must be less than the approximation error.

3. AUTOMATIC PROGRAM

In this section we describe our code *solve int eq,* a program for the automatic solution of Fredholm equations (1.1). Like Atkinson's program *iesimp* the procedure in divided into two stages. In stage A we determine the coarsest meshwidth h_0 by means of (2.4) with the approximate inverse $B_1^{(1)}$. The rate of convergence v_1 is estimated by

$$v_1 = \max_{i=0,1,\ldots,5} (\|f_{1,i+2}-f_{1,i+1}\|/\|f_{1,i+1}-f_{1,i}\|,$$

with $f_{1,0} = T_1 f_0$.

If the rate of convergence v_1 appears sufficiently small such that $w_p^2 < v_p$, then stage B is entered. Otherwise, the number of points N_0 is doubled. In stage B the number of levels is increased until the predicted error estimate for $\|T_p f-f_{p,\sigma}\|$ is less then *tol*. The rate of convergence of the multiple grid process is estimated by w_p (2.5) and the previously determined value of v_1. Using (2.8) we estimate the iteration error by

$$\|f_{p,\sigma}-f_p\| = w_p/(1-w_p)\|f_{p,\sigma}-f_{p,\sigma-1}\|. \tag{3.1}$$

As the number of levels increases we are able to estimate the ratio

$$r = \|T_p f-f_p\|/\|T_{p-1}f-f_{p-1}\|.$$

Asymptotically for $p \to \infty$, this ratio approximates the value $2^{-\alpha}$; see assumption B1 and (2.3). In this paper we only apply multiple grid methods to approximating operators K_p with $\alpha > 1$. Hence, the ratio r must be less than 0.5. Initially we set $r = 0.5$ and we compute the above ratio by

$$r = \min(0.5,\max(2^{-\alpha}, \frac{\|T_{p-1}f_p-f_{p-1}\|}{\|T_{p-2}f_{p-1}-f_{p-2}\|})).$$

Using this value of r we estimate the approximation error by:

$$\|T_p f-f_p\| = r/(1-r)\|T_{p-1}f_{p,\sigma}-f_{p-1}\|. \tag{3.2}$$

The error estimates (3.1) and (3.2) are used to verify the test (2.10). If it is satisfied, then we set $f_p := f_{p,\sigma}$. Otherwise, a new iterate is computed and test (2.10) is repeated. If the error estimate (3.2) yields a value less than *tol*, then the computations are terminated and *solve int eq* returns successfully.

Asymptotically for $p \to \infty$, the total amount of work on level p can be computed. For the examples 1 and 2 the operation counts per iteration sweep for $B_p^{(3)}$ (with $\gamma = 2$) are 3.5 N_p^2. Condition (2.7) insures that only two iterates need to be calculated. The first iterate is obtained by interpolating the results of level p-1. Therefore the total amount of work is equal to:

$$(3.3) \qquad (1+ \frac{1}{4} + \frac{1}{16} + \ldots)\ 3.5\ N_p^2 = 4^{2}/_{3}\ N_p^2.$$

For the description of our code *solve int eq* we use the formal language ALGOL 68 [11]. In practice it appeared that ALGOL 68 is an elegant and in a numerical research environment an efficient tool to implement multiple grid techniques, because this language can easily handle the data structures and the recursive procedures that appear in multiple grid algorithms.

In order to give our program in a concise, modular and readable form, we first give an informal description of a set of ALGOL 68 modes and operators that correspond to the mathematical objects and operators of section 2. The formal description of the modes and operators is their ALGOL 68 - implementation which we give in the appendix.

```
MODE VEC      = REF [ ] REAL:
                 # a structure to represent an element of X_p, i.e. the nodal
                   values of the spline representation #.
MODE MAT      = REF [,] REAL:
                 # a structure to represent a matrix #.
PROC restrict:= (VEC y_p) VEC:
                 # some representation of the operator T_p-1 mapping from
                   X_p onto X_p-1, restrict (y_p) delivers T_p-1 y_p #.
```

```
PROC prologate  := (VEC y_p) VEC:
```

PROC prolongate := (*VEC* y_p) *VEC*:
some representation of the operator T_{p+1} mapping from X_p onto X_{p+1}, *prolongate* (y_p) delivers $T_{p+1}\ y_p$ #.

PROC project := (*INT p*, *PROC* (*REAL*) *REAL f*) *VEC*:
some representation of the operator T_p mapping from X onto X_p, *project* (*p*,*f*) delivers $T_p f$ #.

INT n0 = # an integer to represent the dimension of X_0 #.

PROC n = (*INT p*) *INT*: *n0*2**p*;
delivers the value N_p #.

PROC level = (*INT* n_p) *INT*:
level number as follows from n_p and *n0* #.

PROC zero = (*INT p*) *VEC*:
delivers the zero-element of X_p #.

PROC q := (*INT* ℓ, *VEC* y_p) *VEC*:
some representation of the operator K_p mapping from X_p onto X_ℓ, $q(\ell,y_p)$ delivers $T_\ell K_p y_p$ #.

PROC solve directly = (*MAT a*, *VEC f*,*g*) *VOID*:
solve directly determines the solution of $af = g$ by means of Gaussian-elimination) #.

PROC evaluate jacobian = (*INT p*, *VEC* y_p) *MAT*:
evaluates the matrix $T_p(I-K_p)$ #.

PROC norm = (*VEC* y_p) *REAL*:
delivers the ℓ_∞-norm of y_p #;

PROC kk := (*REAL x*,*y*,*fy*) *REAL*:
some representation the integrand of (1.1),
kk(x,y,fy) = k(x,y)*f(y) #.

PROC forcey := (*REAL x*) *REAL*:
some representation of the right-hand side of 1.1 #.

<u>TEXT 1</u>. An informal description of modes, operators and procedures used in *solve int eq*.

The implementation in an ALGOL 68 program of these operators and procedures depends on the choice of $\{G_p\}$, the approximating operators $\{K_p\}$ and the interpolation operators $\{T_p\}$. Using uniform grids an implementation of example 1 and 2 is given in the appendix.

The approximating inverse $B_p^{(3)}$ is defined in a recursive way and depends on a positive integer γ (fixed for all p). However, a more robust algorithm is obtained if γ can be adapted to the level number p. Therefore, in our implementation of the multiple grid algorithm γ depends on p (i.e. in the definition of $Q_p^{(3)}$ we replace γ by γ_p). Inside *solve int eq* a procedure, called *examine convergence*, is specified which determines γ_p, $p = 1,2,\ldots\ell-1$, (ℓ is the highest level which follows from N_ℓ and N_0). This procedure is of a heuristic structure. In its most simple form it would set $\gamma_p = 2$, for all p. Depending on the behaviour of the actual, iterative process it would adapt γ_p. In order to retain our computational complexity of $O(N^2)$ we take care that $\gamma_p \leq 3$. An exception is made for the lowest level (γ_1 may increase to 5). The multiple grid method obtained in this way is given in text 2.

```
PROC mulgrid3 = ( INT m,sigma,[] INT gamma, MAT jacobian,
                  REF VEC um, VEC rhsm, BOOL um is zero) VOID :
IF   m = 0
THEN solve directly(jacobian,um,rhsm)
ELSE BOOL uz:= um is zero;
     FOR it TO sigma
     DO VEC rm    = ( uz ! rhsm ! rhsm-um+q(m,um));
        VEC umm1:= zero(m-1);
        VEC rmm1  = restrict(rm);
        VEC fmm1  = rmm1-q(m-1,rmm1)+q(m-1,rm);
        mulgrid3(m-1,gamma[m],gamma,jacobian,umm1,fmm1, TRUE );
        um:= um + rm + prolongate(umm1-rmm1);
        uz:= FALSE
     OD
FI   ;
```

<u>TEXT 2</u>. Implementation of the multiple grid method defined by (2.4) and $B_p^{(3)}$.

The procedure *solve int eq* for the automatic solution of Fredholm equations of the second kind is described in text 3. The user has to specify upper limits for N_0 and N_ℓ, i.e. the maximum number of intervals in the coarsest and the finest discretization. Furthermore, *solve int eq* needs information about α (see assumption B3) and $\|T_p\|$ (see 2.5).

```
PROC solve int eq = ( REF INT n0, INT n0upper,n1upper, REAL tol,
                      alfa,normt, REF VEC um, REF REAL error)  BOOL :
BEGIN

  PROC determine v1 = ( REF REAL numv1,denv1,v1, REF VEC um,
                        VEC rhs) VOID :
  ( VEC umold:= COPY um;
    mulgrid3 (1,1,gamma,jacobian,um,rhs, FALSE );
    numv1:= norm(um-umold); REAL v1old:= 0.0;
    FOR  it TO 5
    WHILE umold:= COPY um;
          mulgrid3(1,1,gamma,jacobian,um,rhs, FALSE );
          denv1:= numv1; numv1:= norm(umold-um);
          IF  numv1 > min((tol,1.0e-12))
          THEN (it > 1 ! v1old:= v1 );
               v1:= numv1/denv1;
               it<3 OR ABS (v1-v1old) > 0.02*v1old
          ELSE ( it=1 ! v1*:= ratio ); v1old:=v1; FALSE
          FI
    DO   SKIP   OD ;
    v1:= 1.02*max((v1old,v1))                  );

  PROC examine convergence = ( REAL v1, INT levels, REF [] INT gamma)
        BOOL :
  ( gamma[1]:= 0; BOOL conv;
    FOR  ii TO levels+2
    WHILE IF   ii <= levels
          THEN FOR i FROM 2 TO ii DO gamma[i]:= 3 OD
          ELIF ii=levels+1 THEN gamma[2]:= 4
          ELIF ii=levels+2 THEN gamma[2]:= 5
          FI   ;
          REAL vp:= v1,wp:= v1;
          IF   conv:= ( wp*wp <= vp )
          THEN FOR    p FROM 2 TO levels
               WHILE vp *:= ratio;
                     wp  := vp+wp**gamma[p]*(vp+normt);
                     conv:= ( wp*wp <= vp )
               DO    SKIP    OD
          FI ;
          ( NOT conv) AND n0 = n0upper
    DO   SKIP  OD ;
    print(newline);
    FOR ii TO levels DO print((whole(gamma[ii],4))) OD ;
    conv          );

  PROC v1 on level = ( INT m) REAL :
  ( REAL vp:= v1, wp:= v1;
    FOR p FROM 2 TO m
    DO vp:= ratio*vp; wp:= vp+wp**gamma[p]*(vp+normt) OD ;
    wp      );
```

```
  INT   levels:= level(nlupper); HEAP [1:levels] INT gamma;
  FOR   j TO levels DO gamma[j]:= 2 OD ;
  REAL  ratio = 0.5**alfa;
  REAL  v1:=1.0,denv1,numv1,v2:=0.5,denv2,numv2:=1.0; error:= maxreal;
  BOOL  rapidconvergence;
  VEC   rhs, umm1; MAT jacobian;

#*******************        stage a      ********************************#

  FOR   loop
  WHILE um:= zero(0); rhs:=project(0,forcey);
        jacobian:= evaluate jacobian(0,um);
        solve directly(jacobian,um,rhs);
        IF    loop>1
        THEN  denv2:=numv2; numv2:=norm(restrict(um)-umm1);
              (loop>2 ! v2:= max((ratio,min((0.5,numv2/denv2)))) );
              error:=(v2/(1-v2))*numv2
        FI ;
        IF    error>tol
        THEN  rhs  := project(1,forcey);
              umm1 := COPY um;
              um   := prolongate(umm1);
              determine v1(numv1,denv1,v1,um,rhs);
              rapidconvergence:= examine convergence(v1,levels,gamma)
        FI ;
        error>tol AND ( NOT rapidconvergence) AND n0upper >= 2*n0
  DO    n0*:= 2; levels-:= 1 OD  ;

#******************  end of stage a     ******************************#

  IF    NOT rapidconvergence
  THEN  print((newline," multigrid convergence too slow "))
  FI    ;
```

```
#*******************        stage b       **********************************#

  IF    error > tol AND rapidconvergence
  THEN FOR  m TO levels
       WHILE denv2:=numv2;  numv2:= norm(restrict(um)-umm1);
             REAL rt    := min((ratio,v2));
             REAL um     = v1 on level(m);
             REAL v1m   := um;
             FOR  imax TO 5
             WHILE numv1  > 0.1*((1.0-v1m)/v1m)*(rt/(1.0-rt))*numv2
                    # extra iteration: #
             DO    denv1:= numv1;
                   VEC umold:= COPY um;
                   mulgrid3(m,1,gamma,jacobian,um,rhs, FALSE );
                   numv1:= norm(um-umold);
                   numv2:= norm(restrict(um)-umm1);
                   v1m   := min((um,numv1/denv1))
             OD    ;
             v2    := max((ratio,min((0.5,numv2/denv2))));
             error:= (v2/(1-v2))*numv2;
             error>tol AND m<levels
       DO    rhs := project(m+1,forcey);
             umm1:= COPY um; um:= prolongate(umm1);
             VEC umold:= COPY um;
             mulgrid3(m+1,1,gamma,jacobian,um,rhs, FALSE );
             numv1:= norm(um-umold)
       OD
  FI    ;

#*******************  end of stage b      ******************************#

  error <= tol OR rapidconvergence
END # solve int eq #    ;
```

TEXT 3. Implementation of *solve int eq*.

We could use a slight modification of the above program to implement Atkinson's [1] FORTRAN code *iesimp* in ALGOL 68. Because of the modular structure of *solve int eq* it is also easy to program other numerical methods for integral equations (e.g. higher order formules for smooth kernel functions). Furthermore, *solve int eq* can be applied to multi-dimensional integral equations (e.g. potential flow around three-dimensional bodies) and non-linear integral equations, as was applied in [7] to the physical problem of oscillating disk flow.

4. NUMERICAL RESULTS

In this section we illustrate examples 1-3 on a variety of problems. The problems contain a number of parameters λ,ρ,μ, which are chosen so that $\|(I-T_pK_p)^{-1}\|$ is large. This means that large linear systems are necessary to obtain a reasonably accurate, approximate solution $f_p(s)$ $(\in X_p)$ to the true solution $f(s)$.

For the problems 1-3 (taken from Atkinson [1]) we give the performances of both *solve int eq* and Atkinson's program *iesimp*. In the tables we give the highest level (N_0 and N_ℓ, respectively) and the number of work units (WU), where 1 WU is defined by N_ℓ^2 kernel evaluations.

<u>NOTE</u>. For the examples 1 and 2 of section 2 the number of kernel evaluations is N_ℓ^2, when the values are computed once and stored (example 3 $\approx \frac{4}{3}N^2$). However, when they are not stored the number of kernel evaluations is a good measure for the computational complexity of the algorithms *solve int eq* and *iesimp*.

<u>Case (i):</u>

$$k(x,y) = \begin{cases} -\lambda x(1-y), & 0 \le x \le y \le 1, \\ -\lambda y(1-x), & 0 \le y \le x \le 1. \end{cases}$$

Since this kernel function is not continuously differentiable for $x = y$, we define K_p by the repeated trapezoidal rule. Assumption B3 is satisfied with $\alpha = 2$. The interpolatory spline functions are defined by linear inter-

polation.
The right-hand side g(s) is chosen so that the solution of (1.1) is
$f(s) = \mu^2 s^\mu (1-s)$, $\mu \geq 1$.

NOTE. In [1] Atkinson solves this problem by Simpson's rule, but he remarks that the order of convergence is $O(h^2)$. Therefore, in table I the numerical results of *iesimp* (which are obtained by Simpson's rule) can be compared with the results of *solve int eq*.

	λ	Error		Final		
		Predicted	Actual	N_0	N_ℓ	WU
solve i.	-10.0	8.48 (-4)	7.35 (-4)	32	256	4.83 (1)
iesimp	-10.0	8.93 (-4)	1.03 (-3)	16	256	9.23
solve i.	-30.0	5.52 (-4)	5.48 (-4)	16	128	5.61
iesimp	-30.0	8.91 (-4)	8.93 (-4)	8	128	6.76
solve i.	-90.0	4.85 (-3)	3.86 (-3)	32	256	10.65 (2)
iesimp	-90.0	7.71 (-3)	5.96 (-3)	32	256	13.60 (2)
solve i.	90.0	3.01 (-4)	2.93 (-4)	16	128	8.15 (3)
iesimp	90.0	3.78 (-4)	2.18 (-4)	32	256	8.06 (3)

Table I. Results for case (i), $\mu = 5$.
For *iesimp* K_p is defined by Simpson's rule, for *solve int eq* by the trapezoidal rule.
Tolerance (*tol*) = 1.0 (-3).

Notes to table I:

note (1): For this problem *solve int eq* needs 4.83 WU, which is already close to the asymptotic value of 4.66 WU (see 3.3).

note (2): For the tolerance specified both *iesimp* and *solve int eq* cannot solve this problem with 256 intervals on the highest level. In this case we took *nℓupper* = 256 and therefore both codes fail.

note (3): For this problem *iesimp* needs 256 intervals, whereas *solve int eq*, return succesfully with 128 intervals on the highest level.

Case (ii):

$$k(x,y) = \lambda \cos(\mu^2 \pi xy), \quad 0 \le x, y \le 1.$$

The oscillatory behaviour of this kernel function increases as μ increases. The value of λ is chosen close to characteristic values; see ATKINSON [1]. Since $k(x,y)$ is several times continuously differentiable we use Simpson's rule and cubic interpolation to define K_p and T_p, respectively (i.e. Example 2 of section 2 with $\alpha = \beta = 4$ and $\|T_p\| = 24$).

The right-hand side is so chosen that $f(x) = e^{\mu x} \cos(7\mu x)$.

	λ	μ	Error Predicted	Error Actual	Final N_0	Final N_ℓ	Final WU
solve i.	-2000	1.0	8.41 (-6)	8.37 (-6)	32	256	6.00 (1)
iesimp	-2000	1.0	9.75 (-6)	8.68 (-6)	32	256	7.83
solve i.	-1.42	2.0	3.57 (-6)	3.56 (-6)	16	256	4.17
iesimp	-1.42	2.0	3.58 (-6)	3.57 (-6)	16	256	6.60

Table II. Results for case (ii).
The operator K_p is defined by Simpson's rule.
Tolerance (*tol*) = 1.0 (-5).

note (1): For this case *solve int eq* did not use the default values of γ_p, $(p = 0,1,2)$, but on level 1 the value of γ_p was adapted (i.e. $\gamma_1 = 3$, $\gamma_2 = 2$).

Case (iii):

$$k(x,y) = \mu/[\mu^2+(x-y)^2], \quad \mu > 0, \quad 0 \le x, y \le 1.$$

This kernel is increasingly peaked as $\mu \to 0$. For $\mu = 0.1$ the ratio

$k_{max}/k_{min} = 101$.

We determine the right-hand side so that $f(x) = x^2 - 0.8x + 0.06$. For the definition of K_p and T_p we refer to example 2 of section 2. The numerical results are given in table III.

	λ	Error			Final		
		Tolerance	Predicted	Actual	N_0	N_ℓ	WU
solve i	0.52	1.0 (-7)	3.42 (-8)	3.40 (-8)	32	256	4.83
iesimp	0.52	1.0 (-7)	3.43 (-8)	3.41 (-8)	32	256	7.83
solve i	0.95	1.0 (-6)	1.55 (-7)	1.54 (-7)	32	256	5.22
iesimp	0.95	1.0 (-6)	1.56 (-7)	1.55 (-7)	32	256	7.83
solve i	10.0	1.0 (-6)	1.65 (-7)	1.59 (-7)	32	256	8.14 (1)
iesimp	10.0	1.0 (-6)	2.90 (-7)	2.70 (-7)	32	256	7.83

Table III. Results for case (iii), $\mu = 0.1$.
The operator K_p is defined by Simpson's rule.

note (1): In this problem *solve int eq* adapted the values of γ to: $\gamma_1 = 5$ and $\gamma_2 = 3$. Moreover, an additional iteration sweep was performed on the highest level because test (2.10) was not satisfied after one iteration sweep. Therefore, the computational work deviates from the asymptotic amount given by (3.3).

From the tables I - III we conclude that Atkinson's code *iesimp* is on an average about 50% more expensive than *solve int eq*. In all the experiments both procedures very accurately predict the error of the obtained solution.

Case (iv).

The kernel function is defined in example 3 of section 2. We apply *solve int eq* to the calculation of non-circulatory, potential flow around a Kármán-Trefftz aerofoil, which is obtained from a circle by conformal mapping. For details about the derivation of the integral equation we refer to [2,5,8].

Let the script-letters $\mathcal{X}$, $\mathcal{Z}$, etc. denote complex numbers. The Kármán-Trefftz aerofoil is obtained from the circle in the $\mathcal{X}$-plane, $\mathcal{X} = ce^{i\theta}$, by means of the mapping

$$(4.1) \qquad \mathcal{Z} = f(\mathcal{X}) = (\mathcal{X}-\mathcal{X}_t)^{\mu}/(\mathcal{X}-c(\delta-i\gamma))^{\mu-1},$$

where μ measures the trailing edge angle, γ the camber and δ the thickness of the aerofoil;

$$c = 2\ell(\delta+\sqrt{1-\gamma^2})^{\mu-1}/(2\sqrt{1-\gamma^2})^{\mu}$$

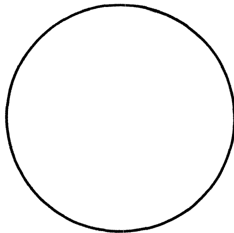
$$\mathcal{X}_t = c(\sqrt{1-\gamma^2}-i\gamma),$$

with ℓ the length of the aerofoil. To make f single-valued we take the principal value in (4.1).

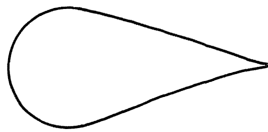
The Kármán-Trefftz aerofoil is not a smooth boundary because of the presence of the trailing edge angle at $\mathcal{Z} = \mathcal{Z}_t$. At this point the curvature is not defined. In the present paper we remove the corner by the additional mapping

$$(4.2) \qquad \mathcal{W} = g(\mathcal{Z}) = (\mathcal{Z}-\mathcal{Z}_t)^{1/\mu}(\mathcal{Z}-\tilde{\mathcal{Z}})^{1-1/\mu},$$

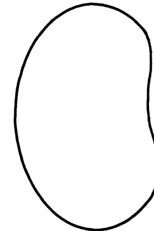
where $\tilde{\mathcal{Z}}$ is a point inside the aerofoil. Here we locate it arbitrarily at $\tilde{\mathcal{Z}} = -\mu$. By means of (4.2) the aerofoil in the $\mathcal{Z}$-plane is converted into a quasi-circular shape in the $\mathcal{W}$-plane. This has been done because the inverse mapping of (4.1) converts real aerofoils (which do not belong to the family of Kármán-Trefftz aerofoils) into quasi-circular shapes too.



$\mathcal{X}$-plane



$\mathcal{Z}$-plane



$\mathcal{W}$-plane

figure 4.1

In the present paper we only consider symmetric aerofoils (γ=0) and non-circulatory flow. The right-hand side of the integral equation is given by

$$g(s) = -2U_\infty \, \underline{\mathrm{re}}\, \omega(s), \qquad \omega \in S.$$

In [9] we also consider non-symmetric aerofoils and circulatory flow.

The grid G_p is given by a uniform partition of the circle. On level p the contour- and collocation points in the ω-plane follow from G_p, (4.1) and (4.2), with

$$G_p = \{\theta_j \mid \theta_j = 2\pi j/N_p, \qquad j = 0,1,\ldots,N_p\}.$$

The tangential velocity v_j at the point $z(s_j)$, ($z \in$ K.T.-aerofoil) is obtained in a numerical way by:

$$(4.3) \qquad v_j = \frac{|f_{p,j+\frac{1}{2}} - f_{p,j-\frac{1}{2}}|}{|\omega_{j+\frac{1}{2}} - \omega_{j-\frac{1}{2}}|} * \left(\frac{d\omega}{dz}\right)\Bigg|_{z=z_j}, \qquad j = 1,\ldots,N_p-1,$$

where $\omega_{j+\frac{1}{2}}$ is the collocation point corresponding with $\theta_{j+\frac{1}{2}}$. In table IV we compare the outcome of (4.3) with the analytical value. The predicted error refers to the error in the approximate solution and not to the error in the tangential velocity.

NOTE. For example 3 the operation counts per iteration sweep (with γ=2) are 3.5 N_p^2. Asymptotically for $p \to \infty$, by condition (2.7) two applications of (2.4) yield an approximation of $\mathcal{O}(h_p)$. In order to obtain an $\mathcal{O}(h_p^{1+\rho})$ approximate solution on the collocation points we have to perform an extra iteration, so that the total amount of work is equal to

$$(4.4) \qquad (1 + \frac{1}{4} + \frac{1}{16} + \ldots)\, 7N_p^2 = 9^{1}/_{3}\, N_p^2.$$

μ	Predicted error in f_p	Actual error in velocity	Final N_0	N_ℓ	WU
1.90	4.23 (-5)	5.42 (-4)	8	256	8.99
1.95	8.60 (-5)	9.09 (-4)	8	256	8.99
1.99	1.84 (-4)	8.49 (-4)	16	256	8.73

Table IV. Results for case (iv).
The operators K_p and T_p are defined in example 3 of section 2.
Flow around a Kármán-Trefftz aerofoil with $\delta = 0.05$, $\gamma = 0.0$, $\ell = 1.0$, $U_\infty = 1.0$.
Tolerance (= *tol*) is 1.0 (-4).

From table IV we conclude that the number of Work Units is in agreement with the asymptotic amount of work (4.4) and our code *solve int eq* appears to be applicable with respect to this aerodynamic problem.

The numerical examples of this section were computed on a CDC-Cyber 175 in single precision arithmetic.

ACKNOWLEDGEMENTS

The author wishes to thank dr. P.W. Hemker for his helpful discussions and drs. P.M. de Zeeuw for his programming assistance and his continuous interest in this work.

REFERENCES

[1] ATKINSON, K.E., *An automatic program for linear Fredholm integral equations of the second kind,* ACM Transactions on Mathematical Software, 2, 1976, pp. 154-171.

[2] BOTTA, E.F.F., *Calculation of potential flow around bodies,* Ph.D. Thesis, Rijksuniversiteit Groningen, 1978.

[3] HACKBUSCH, W., *Die schnelle Auflösung der Fredholmschen Integralgleichung zweiter Art,* Report 78-4, Universität zu Köln (1978).

[4] HEMKER, P.W. & H. SCHIPPERS, *Multiple grid methods for the solution of Fredholm integral equations of the second kind,* Mathematics of Computation (1981).

[5] MARTENSEN, E., *Berechnung der Druckverteilung an Gitterprofilen in ebener Potentialströmung met einer Fredholmschen Integralgleichung,* Arch. Rat. Mech. and Anal. 3, pp. 235-279, 1959.

[6] PRENTER, P.M., *A collocation method for the numerical solution of integral equations,* SIAM J. Numer. Anal 10, (1973), pp. 570-581.

[7] SCHIPPERS, H., *Multiple grid methods for oscillating disk flow* (In: Boundary and Interior Layers - Computational and Asymptotic Methods, J.J.H. Miller, ed., Boole Press, Dublin), 1980.

[8] SCHIPPERS, H., *On the regularity of the principal value of the double layer potential,* Report NW , Mathematisch Centrum, Amsterdam, 1981 (Preprint).

[9] SCHIPPERS, H., *Multiple grid methods for the calculation of potential flow,* to be published.

[10] SLOAN, I.H., E. NOUSSAIR & B.J. BURN, *Projection methods for equations of the second kind,* Journal of Math. Anal. and Appl. 69, (1979), pp. 84-103.

[11] VAN WIJNGAARDEN et al., Eds. (1976), Revised Report *on the Algorithmic Language ALGOL 68,* Springer-Verlag, New York, Heidelberg, Berlin (1976).

APPENDIX

```
BEGIN # automatic solution of fredholm equations of the second kind #

   MODE INTERVAL = STRUCT ( REAL begin,end);
   MODE VEC = REF [ ] REAL ;
   MODE MAT = REF [,] REAL ;

   OP ** = ( REAL x,y) REAL : ( x <= 0.0 ! 0.0 ! exp(y*ln(x)) );

   OP + = ( VEC a,b) VEC :
   ( INT l= LWB a, u= UPB a; VEC c = HEAP [l:u] REAL ;
     FOR i FROM l TO u DO c[i]:= a[i] + b[i] OD ;
     c ) # vec + vec #;

   OP - = ( VEC a,b) VEC :
   ( INT l= LWB a, u= UPB a; VEC c = HEAP [l:u] REAL ;
     FOR i FROM l TO u DO c[i]:= a[i] - b[i] OD ;
     c ) # vec - vec #;

   OP / = ( VEC a, REAL b) VEC :
   ( INT l= LWB a, u= UPB a; VEC c = HEAP [l:u] REAL ;
     FOR i FROM l TO u DO c[i]:= a[i] / b  OD ;
     c ) # vec / real #;

   OP * = ( REAL b, VEC a) VEC :
   ( INT l= LWB a, u= UPB a; VEC c = HEAP [l:u] REAL ;
     FOR i FROM l TO u DO c[i]:= b * a[i] OD ;
     c ) # real * vec #;

   OP * = ( VEC a, b) REAL :
   ( INT l= LWB a, u= UPB a; REAL c:= 0.0;
     FOR i FROM l TO u DO c +:= a[i]*b[i] OD ;
     c ) # vec * vec #;

   OP * = ( MAT a, VEC b) VEC :
   ( INT l=1 LWB a, u=1 UPB a; VEC c = HEAP [l:u] REAL ;
     FOR i FROM l TO u DO c[i] := a[i, ]*b[ ] OD ;
     c ) # mat * vec #;

   OP COPY = ( VEC u) VEC :
   ( INT l = LWB u, up = UPB u; VEC c = HEAP [l:up] REAL ;
     FOR i FROM l TO up DO c[i]:= u[i] OD ;
     c  )  ;

   PROC prvec = ( VEC x) VOID :
   ( print((" vec bounds ", LWB x, UPB x,newline));
     FOR i FROM LWB x TO UPB x
     DO  print(x[i]) OD ;
     print(newline) );
```

```
PROC max = ([] REAL a ) REAL :
( INT l= LWB a, u= UPB a; REAL s:=a[l];
  FOR i FROM l+1 TO u DO ( a[i]>s ! s:=a[i] ) OD ;
  s );

PROC min = ([] REAL a ) REAL :
( INT l= LWB a, u= UPB a; REAL s:=a[l];
  FOR i FROM l+1 TO u DO ( a[i]<s ! s:=a[i] ) OD ;
  s );

PROC norm = ( VEC a) REAL :
( INT l= LWB a, u= UPB a; REAL s:= ABS a[l];
  FOR i FROM l+1 TO u
  DO REAL b = ABS a[i]; ( b>s ! s:=b ) OD ;
  s );

PROC n = ( INT l) INT :( n0 * 2**l );

PROC level = ( INT nb) INT :
( INT  s:= n(0), l:= 0;
  WHILE s < nb DO l+:= 1; s:= 2*s OD ;
  IF   s > nb THEN error FI ; l );

PROC zero = ( INT l) VEC :
( INT nl = n(l); VEC bb = HEAP [0:nl] REAL ;
  FOR i FROM 0 TO nl DO bb[i]:= 0.0 OD ;bb);

PROC pinject = ( INT l, PROC ( REAL ) REAL f) VEC :
( INT nl= n(l);            HEAP [0:nl] REAL yl;
  REAL a:= begin OF int; REAL h = (end OF int - a)/nl;
  yl[0]:= f(a);
  FOR i TO nl
  DO yl[i]:= f( a+:=h ) OD ; yl );

PROC inject = ( VEC vp) VEC :
( INT np = UPB vp, nq = np OVER 2;
  VEC vq = HEAP [0:nq] REAL ;
  FOR i FROM 0 TO nq DO vq[i]:= vp[2*i] OD ; vq);

#*******************      example 1      ********************************#

PROC lin int = ( VEC vp) VEC :
( INT np = UPB vp; INT nq = 2*np;
  VEC vq = HEAP [0:nq] REAL ;
  vq[0]:= vp[0];
  FOR i TO np
  DO  vq[2*i]  := vp[i];
      vq[2*i-1]:= 0.5*(vp[i-1] + vp[i])
  OD  ;
  vq   );
```

```
PROC trap  = (#to level # INT p, VEC v) VEC :
( INT np = n(p), nq =  UPB v;
  VEC vp = HEAP [0:np] REAL ;
  INT st = ( np=nq ! 1 !: 2*np=nq ! 2 ! error; 0 );
  REAL a = begin OF int, b = end OF int; REAL h = (b-a)/nq;
  FOR i FROM 0 BY st TO nq
  DO REAL s, tj; REAL ti = a + i*h;
     s := kk(ti,tj:=a,v[0])/2;
     FOR j TO nq-1
     DO  s +:= kk(ti,tj+:=h,v[j]) OD ;
     s +:=  kk(ti,tj+h,v[nq])/2;
     vp[i OVER st] := s*h
  OD ;
  vp ) ;

#********************     example 2     *********************************#

PROC cub int = ( VEC vp) VEC :
( INT np = UPB vp; INT nq = 2*np;
  VEC vq = HEAP [0:nq] REAL ;
  vq[0]   := vp[0];
  vq[1]   := (5.0*(vp[0]+3.0*vp[1]-vp[2])+vp[3])/16.0;
  vq[nq-2]:= vp[np-1];
  vq[nq-1]:= (5.0*(vp[np]+3.0*vp[np-1]-vp[np-2])+vp[np-3])/16.0;
  vq[nq]  := vp[np];
  FOR i TO np-2
  DO vq[2*i]  := vp[i];
     vq[2*i+1]:= (-vp[i-1]+9.0*(vp[i]+vp[i+1])-vp[i+2])/16.0
  OD ;
  vq   ) ;

PROC simp  = (#to level # INT p, VEC v) VEC :
(  INT np = n(p), nq =  UPB v;
   VEC vp = HEAP [0:np] REAL ;
   INT st = ( np=nq ! 1 !: 2*np=nq ! 2 ! error; 0 );
   REAL a = begin OF int, b = end OF int, w43 = 4/3, w23 = 2/3;
   REAL h = (b-a)/nq;
   FOR i FROM 0 BY st TO nq
   DO REAL s, tj; REAL ti = a + i*h;
      s := kk(ti,tj:=a,v[0])/3;
      FOR j TO nq-1
      DO  s +:= ( ODD j ! w43 ! w23) * kk(ti,tj+:=h,v[j]) OD ;
      s +:=  kk(ti,tj+h,v[nq])/3;
      vp[i OVER st] := s*h
   OD ;
   vp ) ;

#****************** end of  example 2    ****************************#
```

```
PROC solve directly = ( MAT jacobian, VEC um, rhsm) VOID :
BEGIN       # gaussian elimination #
( 1 UPB jacobian /= UPB rhsm OR 2 UPB jacobian/= UPB rhsm !
 error);
MAT jb= jacobian[ AT 1, AT 1];
INT n = UPB jb;
[1:n,1:n+1] REAL a;
VEC v = a[,n+1]; a[,1:n]:= jb; v:= rhsm[ AT 1];
FOR j TO n
DO INT jp1= j+1; INT pj:= j;
   REAL si,s:= ABS a[j,j];
   FOR i FROM jp1 TO n
   DO ((si:= ABS a[i,j]) >s ! s:=si; pj:=i ) OD ;
   IF  j /= pj
   THEN REAL t;
        FOR k TO n+1
        DO t:= a[pj,k]; a[pj,k]:= a[j,k]; a[j,k]:=t OD
   FI  ;
   s := a[j,j];
   FOR i FROM jp1 TO n
   DO si:= a[i,j]/s;
      FOR k FROM j TO n+1
      DO a[i,k] -:= a[j,k]*si OD
   OD
OD ;
FOR j FROM n BY -1 TO 1
DO v[j] /:= a[j,j];
   FOR i FROM j-1 BY -1 TO 1
   DO v[i] -:= a[i,j]*v[j] OD
OD ;
um:= v[ AT (1 LWB jacobian)]
END # solve directly # ;

PROC evaluate jacobian = ( INT m, VEC um ) MAT :
BEGIN INT nm = n(m);        [0:nm] REAL umd := um;
      VEC qm = q(m,um);     HEAP [0:nm,0:nm] REAL jac;
      FOR i FROM 0 TO nm
      DO REAL delta = max(( um[i]*0.001, 0.001));
         umd[i]  +:= delta;
         jac[,i]  := (qm - q(m,umd))/delta;
         jac[i,i]+:= 1.0;
         umd[i]   := um[i]
      OD ; jac
END # evaluate jacobian # ;
```

```
# PROC mulgrid3      =    see text 2                                  #
# PROC solve int eq =     see text 3                                  #

#****************** global variables      ****************************#

INT n0:= 4; REAL tol:=1.0e-6;
INTERVAL int;
PROC ( REAL , REAL , REAL ) REAL kk;
PROC ( REAL ) REAL forcey;
PROC ( INT , PROC ( REAL ) REAL ) VEC project;
PROC ( VEC ) VEC  restrict;
PROC ( VEC ) VEC  prolongate;  REAL normt;
PROC ( INT , VEC ) VEC q ;    REAL alfa ;

#*************   implementation of example 1   ***********************#

 project:=pinject; restrict:=inject;
 prolongate:= lin int;  normt:= 6;  q:=trap;   alfa:=2;

#*************   implementation of example 2   ***********************#

 project:=pinject; restrict:=inject;
 prolongate:= cub int;  normt:= 24;  q:=simp;   alfa:=4;

#*********************************************************************#

#******************  end of library    *****************************#

PR prog PR SKIP
END
```

TEXT 4. An ALGOL 68 program for the automatic solution of Fredholm equations of the second kind, in which the linear system is iteratively solved by a multiple grid method.

Example 1: Approximation of the integral by the trapezoidal rule,

Example 2: Approximation of the integral by Simpson's rule.