stichting
mathematisch
centrum

MC

AFDELING NUMERIEKE WISKUNDE
(DEPARTMENT OF NUMERICAL MATHEMATICS)

NN 20/79 JULI

J. BLOM & J.G. VERWER

AN ALGOL 68 IMPLEMENTATION OF URABE's METHOD FOR NONLINEAR PERIODIC DIFFERENTIAL SYSTEMS

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, 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: 34A50, 65L10

An ALGOL 68 implementation of Urabe's method for nonlinear periodic differential systems

by

J. Blom & J.G. Verwer

ABSTRACT

Let dx/dt = X(x,t) denote a real nonlinear system of differential equations periodic in t. Assuming certain smoothness conditions, for these systems Urabe established a practical method, based on the method of Galerkin, to compute periodic approximate solutions and to obtain strict error bounds. With his method, in the course of calculating the error bounds, it is also possible to investigate the existence of an exact periodic solution, as well as the stability. This report contains a computer implementation of the method of Urabe written in the programming language ALGOL 68. To illustrate the use of the program it is applied to the van der Pol equation and a Volterra-Lotka system.

KEY WORDS & PHRASES: *Nonlinear periodic differential systems, Urabe's method, Numerical software, ALGOL 68.*

1. INTRODUCTION

Let

$$\frac{dx}{dt} = X(x,t) \tag{1.1}$$

denote a real nonlinear system of differential equations where x and X(x,t) are vectors of the same dimension and X(x,t) is smooth and 2π-periodic in t. For this type of differential systems, URABE [3] establishes a practical method, based on the method of Galerkin, to compute 2π-periodic approximate solutions and to obtain strict error bounds. Further he describes a technique to investigate, in the course of calculating error bounds, the existence and stability of an exact periodic solution.

In [4] Urabe and Reiter discuss all numerical aspects which arise in the application of Urabe's method to practical problems. They propose a series of algorithms which lend themselves directly to implementation on a computer. In this paper we present a computer implementation of the method of Urabe, written in the programming language ALGOL 68, which apart from a minor modification, is based on the algorithms given by URABE & REITER [4].

The computer program shall be described in section 2. Because of the fact that almost all programmed algorithms have been given in [4], we shall not discuss the algorithms and confine ourselves, as much as possible, to the ALGOL 68 program. This means, of course, that we shall frequently refer to [4]. It also means that this note can not be read without having knowledge of the paper of Urabe and Reiter. In section 3 we shall, to illustrate the use of the program, apply it to a van der Pol equation [4], and to a Volterra-Lotka system [2]. Other demonstrations of the usefulness of Urabe's method, when implemented on a computer, have been given in [1,4,5].

The program has been tested on the CDC Cyber 73-28/174-16 computer (accuracy: 48 binary digits in the mantissa) of the "Stichting Academisch Rekencentrum Amsterdam".

2. THE ALGOL 68 PROGRAM

The program consists of 3 routines, viz. GALERKIN, COMPUTE M and COMPUTER R, and a prelude. The prelude contains declarations of operators and procedures which perform standard operations, e.g. the linear algebra routines all have been placed in the prelude. It further contains mode definitions, among others:

```
mode vec = ref [ ] real
mode covec = ref [ ] compl
mode mat = ref [ , ] real
mode vecvec = ref [ ] vec
```

Two examples, how to use the routines, will be given in section 3.

2.1. Galerkin

This routine forms the equations of the determining system [4,(2.9)], and computes the vector coefficients a_j, j = 0(1)2m, for the periodic approximation [4,(2.10)]

$$(2.1) \qquad x_m(t) = a_0 + \sum_{n=1}^{m} a_{2n-1} \sin nt + a_{2n} \cos nt$$

to a solution $x = \hat{x}$ of (1.1). The meaning of the parameters of

```
proc  GALERKIN  (int  NC, MS vecvec ALFA,
                 proc (vec,real) vec X,
                 proc (vec,real) mat  PSI) bool
```

is as follows:

NC: The positive integer N of the determining system [4,(2.9)]. The input of NC is a value ≥ m+1.

MS: The integer m in (2.1). The input of MS is some positive integer.

ALFA: The sequence $\alpha = (a_0, a_1, \ldots, a_{2m})$. ALFA[i][j] contains the j-th component of the coefficient vector a_i. The input of ALFA should be some initial approximation for the Newton iteration process [4,(2.12)]. The output of ALFA is the sequence α which defines (2.1).

X: This procedure defines the vector X(x,t) of (1.1). An example of its use is given in section 3.

PSI : This procedure defines the Jacobian matrix $\psi[x,t]$ of $X(x,t)$ with respect to x. An example of its use is given in section 3.

On exit GALERKIN delivers the value true if the computation has been performed successfully, and false if the Newton iteration process did not converge within the maximal number of iterations allowed. When an error occurs in the LU-decomposition of the system [4,(2.12)], the process is broken off. In both cases a message is given by the program (see source text).

```
.proc  galerkin = (.int  nc, ms, .vecvec  alfa,
                   .proc  (.vec , .real ) .vec  x,
                   .proc  (.vec , .real ) .mat  psi) .bool :
( # initialization #

  # genvvec, resp. genmmat reserves heap memory which can be addressed
    by an object of the mode vec, resp. mat,  as well as by an object
    of the mode vecvec, resp. matmat. for further details see the
    description in the prelude.
  #
  .int  nc2 = 2*nc, ms2 = 2*ms;
  .int  upbx = .upb  alfa[ms2];
  .vvec  f = genvvec(ms2, upbx);
  .mmat  jac = genmmat(ms2, upbx);
  .vecvec  xm t = .heap  [1:nc2] .vec ;

  [1:nc2] .real  sin t, cos t;
  .for  i .to  nc2
  .do  .real  ti = (2*i-1)/nc2 * pi;
       sin t[i] := sin(ti); cos t[i] := cos(ti)
  .od ;


  # computation of a trigonometric polynomial.
    see theorem 4 in the paper of urabe and reiter.
  #
  .op  .compute  = (.vecvec  xm t) .void :
  ( .vec  c0, c1, c2, c3, c; .vec  zero = .heap  [1:upbx] .real ;
    .vecvec  alfac = alfa; .int  msc = ms, ms2c = ms2;
    .for  i .to  upbx .do  zero[i] := 0.0 .od ;
    .for  i .to  nc2
    .do  c0 := alfac[ms2c];
         c1 := zero; c2 := zero; c3 := zero;
         .real  cos ti = cos t[i];
         .for  j .from  msc-1 .by  -1 .to  0
         .do  c := alfac[2*j+1] + 2*cos ti*c1 - c3;  c3 := c1; c1 := c;
              c := alfac[2*j]   + 2*cos ti*c0 - c2;  c2 := c0; c0 := c
         .od ;
         xm t[i] := c0 + sin t[i]*c1 - cos ti*c2
    .od
  ); # compute xm #
```

```
# computation of the functions f given by formula (2.9) in the
  paper of urabe and reiter.
#
.op  .compute  = (.vvec  ff) .void :
( .initialize  ff # on zero #;
  .vecvec  f = vv .of  ff;
  .vecvec  xmc t = xm t, alfac = alfa;
  .int  msc = ms, nc2c = nc2, ncc = nc;
  .for  i .to  nc2c
  .do  .real  ti = (2*i-1)/nc2c * pi;
       .vec  xm ti = xmc t[i];
       .vec  x ti  = x(xm ti, ti);
       f[0] +:= x ti;
       .real  sin ti = sin t[i], cos ti = cos t[i];
       .real  sin nti := sin ti, cos nti := cos ti;
       .for  n .to  msc
       .do  f[2*n-1] +:= sin nti * x ti;
            f[2*n]   +:= cos nti * x ti;
            .real  sin = sin nti;
            sin nti := sin*cos ti + cos nti*sin ti;
            cos nti := cos nti*cos ti - sin*sin ti
       .od
  .od ;
  f[0] /:= .real (nc2c);
  .for  n .to  msc
  .do  f[2*n-1] /:= .real (ncc); f[2*n-1] +:= .real (n) * alfac[2*n];
       f[2*n]   /:= .real (ncc); f[2*n]   -:= .real (n) * alfac[2*n-1]
  .od
); # compute f #


# computation of the elements of the jacobian matrix given by
  formula (2.11) in the paper of urabe and reiter.
#
.op  .compute  = (.mmat  jac) .void :
( .initialize  jac # on zero #;
  .matmat  j = mm .of  jac; .vecvec  xmc t = xm t;
  .int  msc = ms, ncc = nc, nc2c = nc2;
  .for  i .to  nc2c
  .do  .real  ti = (2*i-1)/nc2c * pi;
       .vec  xm ti  = xmc t[i]; .mat  psi ti = psi (xm ti, ti);
       j[0,0] +:= psi ti;
       .real  sin ti = sin t[i], cos ti = cos t[i];
       .real  sin pti := sin ti, cos pti := cos ti;
       .for  p .to  msc
       .do  .int  p2 = 2*p; .int  p2m1 = p2-1;
            j[0,p2m1] +:= sin pti * psi ti;
            j[0,p2]   +:= cos pti * psi ti;
            .real  sin nti := sin ti, cos nti := cos ti;
            .for  n .to  msc
            .do  .int  n2 = 2*n; .int  n2m1 = n2-1;
                 j[n2m1,p2m1] +:= sin nti * sin pti * psi ti;
                 j[n2m1,p2]   +:= sin nti * cos pti * psi ti;
                 j[n2,  p2]   +:= cos nti * cos pti * psi ti;
                 .real  sin = sin nti;
                 sin nti := sin*cos ti + cos nti*sin ti;
                 cos nti := cos nti*cos ti - sin*sin ti
            .od ;
            .real  sin = sin pti;
            sin pti := sin*cos ti + cos pti*sin ti;
            cos pti := cos pti*cos ti - sin*sin ti
       .od
  .od ;
```

```
    j[0,0] /:= .real (nc2c);
    .for  p .to  msc
    .do   .int  p2 = 2*p; .int  p2m1 = p2-1;
          .mat (j[p2m1,0]) := j[0,p2m1]; j[0,p2m1] /:= .real (nc2c);
          .mat (j[p2,  0]) := j[0,p2  ]; j[0,p2  ] /:= .real (nc2c)
    .od ;
    .for  n .to  msc
    .do   .int  n2 = 2*n; .int  n2m1 = n2-1;
          j[n2m1,0] /:= .real (ncc); j[n2,0] /:= .real (ncc);
          .for  p .to  msc
          .do   .int  p2 = 2*p; .int  p2m1 = p2-1;
                .mat (j[n2,p2m1]) := j[p2m1,n2];
                j[n2m1,p2m1] /:= .real (ncc);
                j[n2m1,p2  ] /:= .real (ncc);
                j[n2,  p2m1] /:= .real (ncc);
                j[n2,  p2  ] /:= .real (ncc)
          .od ;
          j[n2m1,n2  ] +:= .real (n); j[n2,n2m1] -:= .real (n)
    .od
  ); # compute jac #
```

```
  # newton #

  .real  eps = 1.0 e-11; .int  nonit := 0;
  .while  .compute  xm t;
          .compute  f;
          .compute  jac;
          ( .not  decsol(m .of  jac, v .of  f)
          ! print((newline, 10*"*", " error in lu decomposition"));
            error
          );
          sqrt(.sqr (v .of  f)) > eps  .and  nonit <= 10
  .do  alfa -:= vv .of  f; nonit +:= 1 .od ;
  ( nonit > 10
  ! print((newline, 10*"*", " newton process did not converge within",
          " 10 iterations")); .false
  ! .true
  )
); # galerkin #
```

2.2. Compute M

This routine estimates the upperbound M [4,p.117, section D] and computes the multipliers of $dy/dt = \psi(x_m(t),t)y$, $\psi(x,t)$ being the Jacobian matrix of X(x,t) with respect to x [4,p.118]. The meaning of the parameters of

```
proc COMPUTE M (int LABDA0, MS, vecvec ALFA,
                covec MULTIPLIERS.
                proc (vec,real) mat PSI) real
```

is as follows:

LABDA0 : The integer λ_0 defining the stepsize $h = 2\pi/\lambda_0$ for the Runge-Kutta-method. The input of LABDA0 is some positive integer.

MS : See GALERKIN.

ALFA : See GALERKIN. Here, however, the input of ALFA should be the coefficients of a computed approximation $x_m(t)$. On exit, ALFA is unchanged.

MULTIPLIERS : The multipliers of $dy/dt = \psi(x_m(t),t)y$. No input is required. On exit, MULTIPLIERS contains the computed eigenvalues.

PSI : See GALERKIN.

On exit of COMPUTE M, the estimated upperbound M is assigned to COMPUTE M. It is noted that this routine can be called subsequently for different values of λ_0. This may be desirable in order to get an indication on the accuracy of the results. In case the eigenvalue computation fails the routine gives a message (see source text).

```
.proc  compute m = (.int  labda0, ms, .vecvec  alfa,
                    .ref  .covec  multipliers,
                    .proc  (.vec , .real ) .mat  psi) .real :
( # initialization #

  .int  labda02 = 2*labda0, ms2 = 2*ms;
  .int  upbx = .upb  alfa[ms2];
```

```
# computation of a trigonometric polynomial,
  see theorem 4 in the paper of urabe and reiter.
#
.proc  xm = (.real  t) .vec :
( .vec  c, c0, c1, c2, c3 := .heap  [1:upbx] .real ;
  .vecvec  alfac = alfa; .int  msc = ms, ms2c = ms2;
  .for  j .to  upbx .do  c3[j] := 0.0 .od ;
  c0 := alfac[ms2c];
  c1 := c3; c2 := c3;
  .real  cos t = cos(t), sin t = sin(t);
  .for  j .from  msc-1 .by  -1 .to  0
  .do  c := alfac[2*j+1] + 2*cos t*c1 - c3;  c3 := c1; c1 := c;
       c := alfac[2*j]   + 2*cos t*c0 - c2;  c2 := c0; c0 := c
  .od ;
  c0 + sin t*c1 - cos t*c2
); # compute xm(t) #


# computation of fundamental matrix phi #

.vecmat  phi = .heap  [0:labda0] .mat ;
.for  i .from  0 .to  labda0
.do  phi[i] := .heap  [1:upbx, 1:upbx] .real  .od ;

# runge kutta #
.real  h = pi*2 / labda0; .real  h2 = h/2;
.vec  k0, k1, k2, k3, yn;
.for  j .to  upbx
.do  phi[0][ ,j] := yn := j .unitvec  upbx;
     .mat  psi tn := psi(xm(0), 0), psi tnh2;
     .for  labda .to  labda0
     .do  .real  t = labda * h;
          psi tnh2 := psi(xm(t-h2), t-h2);
          k0 := h * psi tn * yn;
          k1 := h * psi tnh2 * (yn + k0/2.0);
          k2 := h * psi tnh2 * (yn + k1/2.0);
          psi tn := psi(xm(t), t);
          k3 := h * psi tn * (yn + k2);
          phi[labda][ ,j] := yn := yn + (k0 + 2.0*(k1+k2) + k3) / 6.0
     .od
.od ;


# computation of sum of squares of elements of matrix h(t,s) #

.mat  phi 2pi = phi[labda0];
.mat  inv e min phi = .inv  (.unitmat  upbx - phi 2pi);
.vecmat  inv phi = .heap  [0:labda0] .mat ;
.for  i .from  0 .to  labda0
.do  inv phi[i] :=.inv (.heap  [1:upbx,1:upbx] .real  := phi[i]) .od ;

.proc  hc2 = (.int  labda t, labda s) .real :
( .mat  hc =
  ( labda s <= labda t
  ! phi[labda t] * inv e min phi * inv phi[labda s]
  ! phi[labda t] * inv e min phi * phi 2pi * inv phi[labda s]
  );
  .real  sum := 0.0;
  .for  k .to  upbx
  .do  .for  l .to  upbx .do  sum +:= .sqr  hc[k,l] .od  .od ;
  sum
); # hc2 #
```

```
  # actual computation of the upperbound  m #

  .real  mc := 0.0;
  .for  labda t .from  0 .by  2 .to  labda0
  .do  # simpson #
       .real  integral := hc2(labda t, 0) + 4*hc2(labda t, 1) +
                          hc2(labda t, labda0);
       .for  labda s .from  2 .by  2 .to  labda0-2
       .do  integral +:= 2*hc2(labda t, labda s) +
                         4*hc2(labda t, labda s + 1) .od ;
       integral *:= h/3;
       mc := max(mc, integral)
  .od ;
  mc := sqrt(2*pi*mc);


  # computation of multipliers #
  multipliers := eigval (phi 2pi);


  mc
); # compute m #
```

2.3. COMPUTE R

This routine estimates the upperbound r [4,(2.13)]. At this place we deviate from the paper of Urabe and Reiter. Let s(t) be the residual function

$$s(t) = \frac{dx_m(t)}{dt} - X(x_m(t),t), \tag{2.2}$$

and let $t_i = i\pi/P$, i = 0(1)2P-1. We then set, straightforwardly,

$$r = \max_i \| s(t_i) \|. \tag{2.3}$$

The meaning of the parameters of

```
proc COMPUTE R = (int P,MS vecvec ALFA,
                  proc (vec,real) vec X) real
```

is as follows:

P : The grid parameter to determine (2.3). The input of P is some positive integer.

MS : See GALERKIN.

ALFA : See COMPUTE M.

X : See GALERKIN.

On exit of COMPUTE R the estimated value r is assigned to COMPUTE R. It is noted that his routine can be called subsequently for different values of P. This may be desirable in order to get an indication on the accuracy of the result.

```
.proc  compute r = (.int  p, ms, .vecvec  alfa,
                    .proc  (.vec , .real ) .vec  x) .real :
( # initialization #

  .int  p2 = 2*p, ms2 = 2*ms;
  .int  upbx = .upb  alfa[ms2];


  # computation of a trigonometric polynomial,
    see theorem 4 in the paper of urabe and reiter.
  #
  .proc  xm = (.real  t, .vecvec  alfa) .vec :
  ( .vec  c, c0, c1, c2, c3 := .heap  [1:upbx] .real ;
    .int  msc = ms, ms2c = ms2;
    .for  j .to  upbx .do  c3[j] := 0.0 .od ;
    c0 := alfa[ms2c];
    c1 := c3; c2 := c3;
    .real  cos t = cos(t), sin t = sin(t);
    .for  j .from  msc-1 .by  -1 .to  0
    .do  c := alfa[2*j+1] + 2*cos t*c1 - c3;  c3 := c1; c1 := c;
         c := alfa[2*j]   + 2*cos t*c0 - c2;  c2 := c0; c0 := c
    .od ;
    c0 + sin t*c1 - cos t*c2
  ); # compute xm(t) #


  # computation of r #
  # compute coefficients for the derivative of xm #
  .vecvec  alfad := .heap  [0:ms2] .vec ;
  alfad[0] := .heap  [1:upbx] .real ;
  .for  i .to  upbx .do  alfad[0][i] := 0.0 .od ;
  .for  i .to  ms
  .do  alfad[2*i] := .real (i) * alfa[2*i-1];
       alfad[2*i-1] := -.real (i) * alfa[2*i]
  .od ;

  .real  rs := 0;
  .for  i .to  p2
  .do  .real  ti = i*pi / p;
       .real  rsi = sqrt(.sqr (xm (ti, alfad) - x (xm (ti, alfa), ti)));
       rs := max(rsi, rs)
  .od ;
  rs
); # compute r #
```

2.4. The prelude

```
#
prelude of matrix and vector routines
for the computation of periodic solutions to
periodic differential systems.
#


.mode  .vec    = .ref  [] .real ,
       .covec  = .ref  [] .compl ,
       .mat    = .ref  [,] .real ;

.mode  .vecvec  = .ref  [] .vec ,
       .vecmat  = .ref  [] .mat ,
       .matmat  = .ref  [,] .mat ;

.mode  .vvec  = .struct (.vec  v, .vecvec  vv),
       .mmat  = .struct (.mat  m, .matmat  mm);
  # the mode vvec, resp. mmat, is created to permit the addressing
    of a specific part of the heap by an object of the mode vec, resp.
    mat, as well as by an object of the mode vecvec, resp. matmat.
  #
```

```
.proc  genvvec = (.int  ubl, ub2) .vvec :
( .int  width1 = ubl+1, width2 = ub2;

  .vec  v = .heap  [1:width1*width2] .real ;
  .vecvec  vv = .heap  [0:ubl] .vec ;
  .int  index := 0;
  .for  i .from  0 .to  ubl
  .do  vv[i] := v[index+1:index+width2];
       index +:= width2
  .od ;
  (v, vv)
); # genvvec reserves heap memory for the elements of an object of the
     mode vvec, say s.
     v .of  s will contain a reference to [1:(ubl+1)*ub2] .real  and
     vv .of  s a reference to [0:ubl] .vec , where each element consists
     of a reference to [1:ub2] .real .
     @((vv .of  s)[i][j]) = @((v .of  s)[i*ub2+j]).
     ( @: address of)
   #
```

```
.proc  genmmat = (.int  ubl, ub2) .mmat :
( .int  width1 = ubl+1, width2 = ub2;
  .int  width = width1 * width2;

  .mat  m = .heap  [1:width,1:width] .real ;
  .matmat  mm = .heap  [0:ubl,0:ubl] .mat ;
  .int  index1 := 0;
  .for  i .from  0 .to  ubl
  .do  .int  index2 := 0;
       .for  j .from  0 .to  ubl
       .do  mm[i,j] := m[index1+1:index1+width2,
                         index2+1:index2+width2];
            index2 +:= width2
       .od ;
       index1 +:= width2
  .od ;
  (m, mm)
); # genmmat reserves heap memory for the elements of an object of the
     mode mmat, say s.
     m .of  s will contain a reference to
     [1:(ubl+1)*ub2,1:(ubl+1)*ub2] .real  and mm .of  s a reference to
     [0:ubl,0:ubl] .mat , where each element consists of a reference to
     [1:ub2,1:ub2] .real .
     @((mm .of  s)[i,j][k,l]) = @((m .of  s)[i*ub2+k,j*ub2+l]).
     ( @: address of)
   #


.op  .initialize  = (.vvec  vvv) .void :
( .vec  v = v .of  vvv;
  .for  i .to  .upb  v .do  v[i] := 0.0 .od
); # initialize vvec on zero #


.op  .initialize  = (.mmat  mmm) .void :
( .mat  m = m .of  mmm; .int  n = .upb  m;
  .for  i .to  n
  .do  .for  j .to  n
       .do  m[i,j] := 0.0 .od
  .od
); # initialize mmat on zero #



.prio  .unitvec  = 8;
.op  .unitvec  = (.int  i, upb) .vec :
( .vec  e = .heap  [1:upb] .real ;
  .for  j .to  upb .do  e[j] := 0.0 .od ;
  e[i] := 1.0;
  e
); # unitvec #


.op  .unitmat  = (.int  upb) .mat :
( .mat  e = .heap  [1:upb,1:upb] .real ;
  .for  i .to  upb
  .do  .for  j .to  upb .do  e[i,j] := 0.0 .od ;
       e[i,i] := 1.0
  .od ;
  e
); # unitmat #
```

```
.proc  max = (.real  a, b) .real :
( a > b! a! b );


.op  .sqr  = (.real  x) .real : x*x;


.op  .sqr  = (.vec  v) .real :
( .real  norm := 0.0;
  .for  i .to  .upb v .do  norm +:= .sqr  v[i] .od ;
  norm
);




.op  +:= = (.vec  v1, v2) .vec :
( .for  i .to  .upb  v1
  .do  v1[i] +:= v2[i] .od ;
  v1
); # vec +:= vec #


.op  + = (.vec  v1, v2) .vec :
( .int  n = .upb v1; .vec  v = .heap  [1:n] .real ;
  .for  i .to  n .do  v[i] := v1[i] + v2[i] .od ;
  v
); # vec + vec #


.op  +:= = (.mat  m, .real  r) .mat :
( .for  i .to  .upb  m
  .do  m[i,i] +:= r .od ;
  m
); # mat +:= real * unit mat #


.op  +:= = (.mat  m1, m2) .mat :
( .int  n = .upb  m1;
  .for  i .to  n
  .do  .for  j .to  n
       .do  m1[i,j] +:= m2[i,j] .od
  .od ;
  m1
); # mat +:= mat #




.op  -:= = (.vec  v1, v2) .vec :
( .for  i .to  .upb  v1
  .do  v1[i] -:= v2[i] .od ;
  v1
); # vec -:= vec #


.op  - = (.vec  v1, v2) .vec :
( .int  n = .upb v1; .vec  v = .heap  [1:n] .real ;
  .for  i .to  n .do  v[i] := v1[i] - v2[i] .od ;
  v
); # vec - vec #
```

```
.op  -:= = (.mat  m, .real  r) .mat :
( .for  i .to  .upb  m
  .do  m[i,i] -:= r .od ;
  m
); # mat -:= real * unit mat #


.op  - = (.mat  m1, m2) .mat :
( .int  n = .upb m1; .mat  m = .heap  [1:n,1:n] .real ;
  .for  i .to  n
  .do  .for  j .to  n .do  m[i,j] := m1[i,j] - m2[i,j] .od  .od ;
  m
); # mat - mat #


.op  -:= = (.vecvec  vv1, .vecvec  vv2) .vecvec :
( .int  n1 = .upb  vv1; .int  n2 = .upb  vv1[n1];
  .for  i .from  0 .to  n1
  .do  .for  j .to  n2
       .do  vv1[i][j] -:= vv2[i][j] .od
  .od ;
  vv1
); # vecvec -:= vecvec #




.op  * = (.real  r, .vec  v) .vec :
( .int  n = .upb  v;
  .vec  vc = .heap  [1:n] .real ;
  .for  i .to  n
  .do  vc[i] := r * v[i] .od ;
  vc
); # real * vec #


.op  * = (.vec  v1, v2) .real :
( .real  inprod := 0.0;
  .for  i .to  .upb v1 .do  inprod +:= v1[i] * v2[i] .od ;
  inprod
); # innerproduct of vectors #


.op  * = (.real  r, .mat  m) .mat :
( .int  n = .upb  m;
  .mat  rm = .heap  [1:n,1:n] .real ;
  .for  i .to  n
  .do  .for  j .to  n
       .do  rm[i,j] := r * m[i,j] .od
  .od ;
  rm
); # real * mat #
```

```
.op  * = (.mat  m, .vec  v) .vec :
( .int  n = .upb  v;
  .vec  mv = .heap  [1:n] .real ;
  .for  i .to  n
  .do  .real  s := 0.0;
       .for  j .to  n
       .do  s +:= m[i,j] * v[j] .od ;
       mv[i] := s
  .od ;
  mv
); # mat * vec #


.op  * = (.mat  m1, m2) .mat :
( .int  n = .upb  m1;
  .mat  m = .heap  [1:n,1:n] .real ;
  .for  j .to  n
  .do  m[ ,j] := m1 * m2[ ,j] .od ;
  m
); # mat * mat #




.op  /:= = (.vec  v, .real  r) .vec :
( .for  i .to  .upb  v
  .do  v[i] /:= r .od ;
  v
); # vec /:= real #


.op  / = (.vec  v, .real  r) .vec :
( .int  n = .upb  v;
  .vec  w = .heap  [1:n] .real ;
  .for  i .to  n
  .do  w[i] := v[i]/r .od ;
  w
); # vec / real #


.op  /:= = (.mat  m, .real  r) .mat :
( .int  n = .upb  m;
  .for  i .to  n
  .do  .for  j .to  n
       .do  m[i,j] /:= r .od
  .od ;
  m
); # mat /:= real #
```

```
.proc  eigval = (.mat  m) .covec :
( # computation of complex eigenvalues of a square matrix.
    first the matrix is transformed to hessenberg form, next a schur
    decomposition is performed with mixed single and double shift.
    ( see louter-nool, m., the calculation of eigenvalues and invariant
    subspaces, report nw, mathematisch centrum, amsterdam (to appear) ).
    in case the order of  m  is less than three the eigenvalues are
    directly computed.
  #
  .int  n = .upb m; .covec  eigval = .heap  [1:n] .compl ;
  ( n = 1
  ! eigval[1] := m[1,1]
  !:n = 2
  ! .real  m11 = m[1,1], m12 = m[1,2], m21 = m[2,1], m22 = m[2,2];
    .real  m11pm22 = m11 + m22;
    .real  discr = .sqr  m11pm22 - 4*(m11*m22-m12*m21);
    ( discr >= 0
    ! .real  sqrt discr = sqrt(discr);
      eigval[1] := ((m11pm22 + sqrt discr)/2, 0.0);
      eigval[2] := ((m11pm22 - sqrt discr)/2, 0.0)
    ! .real  sqrt discr = sqrt(-discr);
      eigval[1] := (m11pm22/2, sqrt discr/2);
      eigval[2] := (m11pm22/2,-sqrt discr/2)
    )
  ! hessenberg (m);
    ( .int  nev = schur (m, eigval); nev > 0
    ! print((newline, 10*" ", " only", whole(n-nev,-3), " eigenvalues ",
            "are found"));
      .for  i .to  nev .do  eigval[i] := (0.0, 0.0) .od
    )
  );
  eigval
); # eigval #



.proc  hessenberg = (.mat  m) .void :
( .int  n = .upb m; .real  norm := 0.0;
  .for  i .to  n
  .do  .for  j .to  n
       .do  norm := max (.abs m[i,j], norm) .od
  .od ;
  .for  k .to  n-2
  .do  .int  kpl = k+1; .real  maxi := 0.0;
       .for  i .from  n .by  -1 .to  k+2
       .do  maxi := max (.abs m[i,k], maxi) .od ;
       ( maxi >= small real * norm
       ! maxi := max (.abs m[kpl,k], maxi);
         m[kpl:n,k] /:= maxi;
         .real  sigma = sqrt(.sqr m[kpl:n,k]) * ( m[kpl,k] < 0! -1! 1 );
         .real  x = m[kpl,k] +:= sigma;
         .real  pik = sigma * x;
         .for  j .from  kpl .to  n
         .do  m[kpl:n,j]-:=m[kpl:n,k]*m[kpl:n,j]/pik * m[kpl:n,k] .od ;
         .for  i .to  n
         .do  m[i,kpl:n]-:=m[i,kpl:n]*m[kpl:n,k]/pik * m[kpl:n,k] .od ;
         m[kpl,k] := -maxi * sigma
       )
  .od
); # hessenberg #
```

```
.proc  schur = (.mat  m, .covec  eig) .int :
( .int  count; .int  n = .upb m; .int  maxit = 10*n;
  .real  tol = 1.0 e-13;.real  correction = 1.0 e-7;


  .proc  rot2 = (.ref .real  c, s) .real :
  ( .real  maxi = max (.abs c, .abs s);
    c /:= maxi; s /:= maxi;
    .real  delta = sqrt (c*c + s*s);
    c /:= delta; s /:= delta;
    maxi * delta
  );


  .proc  rot3 = (.ref .real  a,b,c,pi) .real :
  ( .real  maxi = max (.abs a, max(.abs b, .abs c));
    a /:= maxi; b /:= maxi; c /:= maxi;
    .real  sigma = sqrt (a*a + b*b + c*c) * ( a < 0! -1! 1 );
    a +:= sigma; pi := a * sigma;
    maxi * sigma
  );


  .proc  qrsingle = (.int  l, u, .real  shift) .void :
  ( .vec  tau; .real  cl, sl;
    m[l,l] -:= shift;
    .for  k .from  l .to  u-1
    .do  .int  kpl = k+1;  m[kpl,kpl] -:= shift;
         .real  c := m[k,k], s := m[kpl,k];
         .real  rot := rot2 (c, s);
         ( k > l! m[k,k-1] := rot*sl; rot*:=cl );
         m[k,k]:=rot;
         tau := .heap  [l:u-k] .real  := m[k,kpl:u];
         m[k,kpl:u] := c*tau + s*m[kpl,kpl:u];
         m[kpl,kpl:u] := c*m[kpl,kpl:u] - s*tau;
         tau := .heap  [l:kpl-1] .real  := m[l:k,k];
         m[l:k,k] := c*tau + s*m[l:k,kpl];
         m[l:k,kpl] := c*m[l:k,kpl] - s*tau;
         m[k,k] +:= shift;
         cl := c; sl := s
    .od ;
    m[u,u-1] := sl*m[u,u]; m[u,u] := cl*m[u,u] + shift;
    count +:= 1
  ); # qrsingle #
```

```
.proc  qrdouble = (.int  l, u, .real  shift) .void :
( .int  lpl = l + 1, uml = u - 1; .vec  tau;
  .for  k .from  l-1 .to  u-3
  .do  .int  kpl = k + 1, kp2 = k + 2, kp3 = k + 3, kp4 = k + 4;
       .real  ml, m2, m3;
       ( k >= l
       ! ml := m[kpl,k]; m2 := m[kp2,k]; m3 := m[kp3,k]
       ! m[l,l] -:= shift; m[lpl,lpl] -:= shift;
         m2 := m[uml,uml] - shift; m3 := -m[u,uml] * m[uml,u];
         ml := (m[l,l] * (m[l,l]-m2) + m3) / m[lpl,l] + m[l,lpl];
         m2 := m[l,l] + m[lpl,lpl] -m2;
         m3 := m[l+2,lpl]
       );
       .real  pi;
       .real  rot = rot3 (ml, m2, m3, pi);
       ( k < l! m[kp3,kpl] := 0.0! m[kpl,k] := -rot );
       m[kp3,kp3] -:= shift;
       tau:= (ml*m[kpl,kpl:u] + m2*m[kp2,kpl:u] + m3*m[kp3,kpl:u])/pi;
       m[kpl,kpl:u] -:= ml * tau;
       m[kp2,kpl:u] -:= m2 * tau;
       m[kp3,kpl:u] -:= m3 * tau;

       .int  min = ( kp4 > u! u! m[kp4,kpl] := 0; m[kp4,kp2] := 0;
                                 kp4 );
       tau :=(ml*m[l:min,kpl] + m2*m[l:min,kp2] + m3*m[l:min,kp3])/pi;
       m[l:min,kpl] -:= ml * tau;
       m[l:min,kp2] -:= m2 * tau;
       m[l:min,kp3] -:= m3 * tau;
       m[kpl,kpl] +:= shift
  .od ;
  .real  ml := m[uml,u-2], m2 := m[u,u-2];
  m[uml,u-2] := rot2 (ml, m2);
  tau := .heap  [1:2] .real  := m[uml,uml:u];
  m[uml,uml:u] := ml*tau + m2*m[u,uml:u];
  m[u,uml:u] := ml*m[u,uml:u] - m2*tau;
  tau := .heap  [1:u-l+1] .real  := m[l:u,uml];
  m[l:u,uml] := ml*tau + m2*m[l:u,u];
  m[l:u,u] := ml*m[l:u,u] - m2*tau;
  m[uml,uml] +:= snift; m[u,u] +:= shift;
  count +:= 1
); # qrdouble #



.proc  deflation = (.int  u) .int :
( .int  bg := u; .bool  b := .true ;
  .while  b .and  bg > 1
  .do  ( .abs  m[bg,bg-1] > tol! bg -:= 1
                              ! b := .false ; m[bg,bg-1] := 0 )
  .od ;
  bg
); # deflation #
```

```
  # start of schur #
  .int  og := n; count := 0;
  .while  og > 0 .and  count < maxit
  .do  .int  bg = deflation (og); .int  ogml = og -1;
       ( .int  l = og - bg; l = 0
       ! eig[og] := (m[og,og], 0.0); og := ogml
       !:.real  corr =
                ( l > l
                ! sqrt(correction*.abs (m[og,ogml]*m[ogml,og-2]))
                ! 0.0
                );
         .real  delta := (m[ogml,ogml] - m[og,og])/2;
         .real  det = m[og,ogml] * m[ogml,og];
         .real  discr = delta * delta + det;
         discr < 0
       ! ( l = l
         ! .real  re = delta + m[og,og], im = sqrt(-discr);
           eig[og]   := (re, im);
           eig[ogml] := (re,-im);
           og -:= 2
         ! qrdouble (bg, og, m[og,og]+corr)
         )
       !:.real  s;
         ( .abs delta >= tol
         ! delta := l/delta;
           s := -delta * det/(sqrt(.sqr delta*det+l) + l)
         !:det < .sqr delta! s := 0! s := sqrt(det)
         );
         l = l
       ! eig[og]   := (m[og,og] + s, 0.0);
         eig[ogml] := (m[ogml,ogml] - s, 0.0);
         og -:= 2
       ! qrsingle (bg, og, m[og,og]+s+corr)
       )
  .od ;
  og
); # schur #
```

```
.op  .inv  = (.mat  m) .mat :
( # inversion of a square matrix using lu - decomposition.
    in case the order of  m  is less than two the computation is
    performed directly.
  #
  .int  n = .upb  m;
  n = l
! ( .real  x = m[l,l]; .abs x < small real * x*x
  ! print((newline, "singular matrix")); error; m
  ! m[l,l] := l/x; m
  )
! [l:n] .int  p;
  ( .not  dec(m,p)! print((newline, "error in lu decomp"));error );
  .mat  inv = .unitmat  n;
  .for  j .to  n
  .do  sol(m, p, inv[ ,j]) .od ;
  inv
); # inverse of matrix #
```

```
.proc  decsol = (.mat  a, .vec  b) .bool :
( # solution of linear system by means of lu - decomposition with
    partial pivoting.
    incase the order of  a  is less than three the computation is
    performed directly.
  #
  .int  n = .upb b;
  n = 1
! ( .real  x = a[1,1]; .abs x <= small real * x*x
  ! .false
  ! b[1] /:= x; .true
  )
!:n = 2
! .real  a11 := a[1,1], a12 := a[1,2],
         a21 := a[2,1], a22 := a[2,2],
         b1  := b[1],   b2  := b[2];
  .real  eps = small real * max (a11*a11+a12*a12, a21*a21+a22*a22);
  ( .abs a11 < .abs a21
  ! .real  x := a11; a11 := a21; a21 := x;
           x := a12; a12 := a22; a22 := x;
           x := b1;  b1  := b2;  b2  := x
  );
  ( .abs a11 < eps
  ! .false
  !:.real  x = a21/a11; .real  y = a22 - a12*x;
    .abs y < eps
  ! .false
  ! b2 := b[2] := (b2 - x*b1) / y;
    b[1] := (b1 - a12*b2) / a11;
    .true
  )
! [1:n] .int  p;
  ( dec (a, p)! sol (a, p, b); .true ! .false  )
); # decsol #
```

```
.proc  dec = (.mat  a, .ref  [] .int  p) .bool :
( # lu - decomposition of a square matrix #
  .int  n = .upb p, .bool  notsing := .true ;
  [1:n] .real  v; .real  r := -1;
  .for  i .to  n
  .do  .real  s = .sqr  a[i, ];
       r := max (r, s);
       v[i] := 1/s
  .od ;
  .real  eps = small real * r;
  .for  k .to  n .while  notsing
  .do  .real  max := 0.0, .int  pk := k, .vec  colk = a[ ,k];
       .for  i .from  k .to  n
       .do  ( .real  s = .abs (colk[i] -:= a[i, :k-1] * colk[ :k-1])
                         * v[i];
              s > max! pk := i; max := s )
       .od ;
       ( max < eps
       ! notsing := .false
       ! .vec  rowk = a[k, ];
         p[k] := pk;
         ( pk /= k! [1:n] .real  h := a[pk, ];
                    a[pk, ] := rowk; rowk := h; v[pk] := v[k] );
         .for  i .from  k+1 .to  n
         .do  rowk[i] -:= rowk[ :k-1] * a[ :k-1,i] .od ;
         rowk[k+1: ] := 1/rowk[k] * rowk[k+1: ]
       )
  .od ;
  notsing
); # dec #
```

```
.proc  sol = (.mat  a, .ref  [] .int  p, .vec  b) .void :
( # solution of linear system where the matrix is in lu - form #
  .int  n = .upb p;
  .for  k .to  n
  .do  .int  pk = p[k], .real  r = b[k];
       b[k] :=(b[pk] - a[k, :k-1]*b[ :k-1]) / a[k,k];
       ( pk /= k! b[pk] := r )
  .od ;
  .for  k .from  n-1 .by  -1 .to  1
  .do  b[k] -:= a[k,k+1: ] * b[k+1: ] .od
); # sol #
```

3. TWO EXAMPLES

We shall give two examples to show how one can use the subroutines. For each example we list the ALGOL 68 program and its output.

3.1. The van der Pol equation

Our first example is the van der Pol equation considered in [4, example 3, p.133]:

$$\frac{dx}{dt} = y,$$

$$\frac{dy}{dt} = -x + 0.1(1-x^2)y + 0.1 \sin t. \tag{3.1}$$

For a specification of the various parameters, see the program text.

```
example  urabe:
( # procedure to print the computed galerkin approximation #

  .proc  print xmt = (.vecvec  alfa) .void :
  (
    .proc  printalfa = (.real  alfa) .void :
     print((" ( ", fixed(alfa, 12, 9), " )"));

    print((newpage, "xm(t) =  "));
    .int  upbx = .upb  alfa[0], ms = .upb  alfa .over  2;
    .for  j .to  upbx
    .do  printalfa(alfa[0][j]); print((newline, 9*" ")) .od ;
    .for  i .to  ms
    .do  .vec  alfasin = alfa[2*i-1], alfacos = alfa[2*i];
         print((newline, 8*" ","+")); printalfa(alfasin[1]);
         print((" . sin", whole(i,-2),"t +"));
         printalfa(alfacos[1]);
         print((" . cos", whole(i,-2), "t", newline));
         .for  j .from  2 .to  upbx
         .do  print(9*" "); printalfa(alfasin[j]); print(11*" ");
              printalfa(alfacos[j]); print(newline)
         .od
    .od
  ); # print xm(t) #



  # right hand side function of the problem #

  .proc  x = (.vec  xm t, .real  t) .vec :
  ( .real  x = xm t[1], y = xm t[2];
    .heap  [1:2] .real  :=
    (y, -x + 0.1*(1-x*x)*y + 0.1*sin(t))
  );
```

```
  # jacobian of the right hand side function #

  .proc  psi = (.vec  xm t, .real  t) .mat :
  ( .real  x = xm t[1], y = xm t[2];
    .heap  [1:2,1:2] .real  :=
    ((0.0           , 1.0        ),
     (-1.0 - 0.2*x*y, 0.1*(1-x*x)))
  );



  # computation of the galerkin approximation of order 15 #

  .int  nc = 32, ms = 15;
  .vecvec  alfa := .heap  [0:2*ms] .vec ;
  # initial approximation #
  .for  i .from  0 .to  2*ms
  .do  alfa[i] := .heap  [1:2] .real  := (0.0, 0.0) .od ;
  alfa[1][1] :=-0.1423; alfa[2][1] :=-2.3788;
  alfa[1][2] := 2.3788; alfa[2][2] :=-0.1423;

  galerkin ( nc, ms, alfa, x, psi);
  print xmt (alfa);


  # computation of tne upperbound  m  and the multipliers.
    (for three values of labda0)
  #

  .covec  multipliers; # ms = 15 #
  print((newline, newline, "labda0",13*" ", "m", 26*" ", "multipliers",
        newline));
  .for  i .to  3
  .do  .int  labda0 = ( i! 64, 128, 256 );
       .real  mc = compute m (labda0, ms, alfa, multipliers, psi);
       print((whole(labda0,-4), 7*" ", float(mc, 16, 9, 3), 5*" ",
              float(.re multipliers[1], 16, 9, 3), 3*" ",
              float(.im multipliers[1], 16, 9, 3), "  i",newline,32*" ",
              float(.re multipliers[2], 16, 9, 3), 3*" ",
              float(.im multipliers[2], 16, 9, 3), "  i",newline,
              newline))
  .od ;


  # computation of the residual constant r. (for three values of p)
  #

  # ms = 15 #
  print((newline, newline, "  p   ",13*" ", "r",  newline));
  .for  i .to  3
  .do  .int  p = ( i! 16, 32, 64 );
       .real  rs = compute r (p, ms, alfa, x);
       print((whole(p,-4), 7*" ", float(rs, 16, 9, 3), newline))
  .od
)
```

```
xm(t) =    ( -0.000000000 )
           ( +0.000000000 )

         + ( -0.142330101 ) . sin 1t + ( -2.378785902 ) . cos 1t
           ( +2.378785902 )            ( -0.142330101 )

         + ( +0.000000000 ) . sin 2t + ( +0.000000000 ) . cos 2t
           ( -0.000000000 )            ( +0.000000000 )

         + ( +0.041867539 ) . sin 3t + ( -0.004646924 ) . cos 3t
           ( +0.013940772 )            ( +0.125602617 )

         + ( +0.000000000 ) . sin 4t + ( +0.000000000 ) . cos 4t
           ( -0.000000000 )            ( +0.000000000 )

         + ( +0.000215279 ) . sin 5t + ( +0.001223706 ) . cos 5t
           ( -0.006118531 )            ( +0.001076393 )

         + ( -0.000000000 ) . sin 6t + ( +0.000000000 ) . cos 6t
           ( -0.000000000 )            ( +0.000000000 )

         + ( -0.000039873 ) . sin 7t + ( +0.000009756 ) . cos 7t


         + ( +0.000000000 ) . sin 8t + ( +0.000000000 ) . cos 8t
           ( -0.000000000 )            ( +0.000000000 )

         + ( -0.000000430 ) . sin 9t + ( -0.000001358 ) . cos 9t
           ( +0.000012223 )            ( -0.000003869 )

         + ( +0.000000000 ) . sin10t + ( +0.000000000 ) . cos10t
           ( -0.000000000 )            ( -0.000000000 )

         + ( +0.000000047 ) . sin11t + ( -0.000000019 ) . cos11t
           ( +0.000000204 )            ( +0.000000521 )

         + ( +0.000000000 ) . sin12t + ( +0.000000000 ) . cos12t
           ( -0.000000000 )            ( -0.000000000 )

         + ( +0.000000001 ) . sin13t + ( +0.000000002 ) . cos13t
           ( -0.000000022 )            ( +0.000000010 )

         + ( +0.000000000 ) . sin14t + ( -0.000000000 ) . cos14t
           ( -0.000000000 )            ( -0.000000000 )

         + ( -0.000000000 ) . sin15t + ( +0.000000000 ) . cos15t
           ( -0.000000001 )            ( -0.000000001 )
```

lambda0	m	multipliers	
64	+5.706754181e +1	+8.761171414e -1	+0.000000000e +0 i
		+3.591361143e -1	+0.000000000e +0 i
128	+5.712478531e +1	+8.761186966e -1	+0.000000000e +0 i
		+3.591344918e -1	+0.000000000e +0 i
256	+5.716251221e +1	+8.761187707e -1	+0.000000000e +0 i
		+3.591343828e -1	+0.000000000e +0 i

p	r
16	+6.996856587e-10
32	+7.489440616e-10
64	+7.489440616e-10

3.2. A Volterra-Lotka system

Our second example belongs to the class of Volterra-Lotka systems discussed in [2]:

$$\frac{dx}{dt} = (1+0.4\cos t)x - xy - 0.9\,x^2, \qquad \frac{dy}{dt} = -y + xy. \tag{3.2}$$

For a specification of the various parameters, see the program text.

```
volterra lotka:
( # procedure to print the computed galerkin approximation #

  .proc  print xmt = (.vecvec  alfa) .void :
  (
    .proc  printalfa = (.real  alfa) .void :
     print((" ( ", fixed(alfa, 12, 9), " )"));

    print((newpage, "xm(t) =  "));
    .int  upbx = .upb  alfa[0], ms = .upb  alfa .over  2;
    .for  j .to  upbx
    .do  printalfa(alfa[0][j]); print((newline, 9*" ")) .od ;
    .for  i .to  ms
    .do  .vec  alfasin = alfa[2*i-1], alfacos = alfa[2*i];
         print((newline, 6*" ","+")); printalfa(alfasin[1]);
         print((" . sin", whole(i,-2),"t +"));
         printalfa(alfacos[1]);
         print((" . cos", whole(i,-2), "t", newline));
         .for  j .from  2 .to  upbx
         .do  print(9*" "); printalfa(alfasin[j]); print(11*" ");
              printalfa(alfacos[j]); print(newline)
         .od
    .od
  ); # print xm(t) #



  # right hand side function of the problem #

  .proc  x = (.vec  xm t, .real  t) .vec :
  ( .real  x = xm t[1], y = xm t[2];
    .heap  [1:2] .real  :=
    ((1+0.4*cos(t))*x - x*y -0.9*x*x, -y + x*y)
  );


  # jacobian of the right hand side function #

  .proc  psi = (.vec  xm t, .real  t) .mat :
  ( .real  x = xm t[1], y = xm t[2];
    .heap  [1:2,1:2] .real  :=
    ((1 + 0.4*cos(t) - y - 1.8*x, -x     ),
     (y                         , -1 + x))
  );
```

```
  # computation of the galerkin approximation of order 15 #

  .int  nc = 32, ms = 15;
  .vecvec  alfa := .heap  [0:2*ms] .vec ;
  # initial approximation #
  alfa[0] := .heap  [1:2] .real  := (1.0  , 0.1 );
  alfa[1] := .heap  [1:2] .real  := (0.22 , 0.04);
  alfa[2] := .heap  [1:2] .real  := (0.22 ,-0.04);
  .for  i .from  3 .to  2*ms
  .do  alfa[i] := .heap  [1:2] .real  := (0.0, 0.0) .od ;

  galerkin ( nc, ms, alfa, x, psi);
  print xmt (alfa);


  # computation of the upperbound  m  and the multipliers.
    (for three values of labda0)
  #

  .covec  multipliers; # ms = 15 #
  print((newline, newline, "labda0",13*" ", "m", 26*" ", "multipliers",
        newline));
  .for  i .to  3
  .do  .int  labda0 = ( 1! 64, 128, 256 );
       .real  mc = compute m (labda0, ms, alfa, multipliers, psi);
       print((whole(labda0,-4), 7*" ", float(mc, 16, 9, 3), 5*" ",
              float(.re multipliers[1], 16, 9, 3), 3*" ",
              float(.im multipliers[1], 16, 9, 3), "  i",newline,32*" ",
              float(.re multipliers[2], 16, 9, 3), 3*" ",
              float(.im multipliers[2], 16, 9, 3), "  i",newline,
              newline))
  .od ;


  # computation of the residual constant r. (for three values of p)
  #

  # ms = 15 #
  print((newline, newline, "  p   ",13*" ", "r",  newline));
  .for  i .to  3
  .do  .int  p = ( 1! 16, 32, 64 );
       .real  rs = compute r (p, ms, alfa, x);
       print((whole(p,-4), 7*" ", float(rs, 16, 9, 3), newline))
  .od
)
```

```
xm(t) =    ( +1.000000000 )
           ( +0.100000000 )

         + ( +0.221021961 ) . sin 1t + ( +0.218472259 ) . cos 1t
           ( +0.021657960 )            ( -0.021681436 )

         + ( +0.021225670 ) . sin 2t + ( +0.008086503 ) . cos 2t
           ( -0.001969994 )            ( -0.001026719 )

         + ( +0.001231897 ) . sin 3t + ( +0.000702737 ) . cos 3t
           ( -0.000046970 )            ( +0.000116485 )

         + ( +0.000081258 ) . sin 4t + ( +0.000067509 ) . cos 4t
           ( +0.000005602 )            ( +0.000002214 )

         + ( +0.000005469 ) . sin 5t + ( +0.000009002 ) . cos 5t
           ( +0.000000102 )            ( -0.000000236 )

         + ( +0.000000313 ) . sin 6t + ( +0.000000845 ) . cos 6t
           ( -0.000000009 )            ( -0.000000005 )

         + ( +0.000000011 ) . sin 7t + ( +0.000000076 ) . cos 7t
           ( -0.000000000 )            ( +0.000000000 )

         + ( -0.000000000 ) . sin 8t + ( +0.000000007 ) . cos 8t
           ( +0.000000000 )            ( +0.000000000 )

         + ( -0.000000000 ) . sin 9t + ( +0.000000001 ) . cos 9t
           ( +0.000000000 )            ( -0.000000000 )

         + ( -0.000000000 ) . sin10t + ( +0.000000000 ) . cos10t
           ( -0.000000000 )            ( -0.000000000 )

         + ( -0.000000000 ) . sin11t + ( +0.000000000 ) . cos11t
           ( -0.000000000 )            ( +0.000000000 )

         + ( -0.000000000 ) . sin12t + ( +0.000000000 ) . cos12t
           ( -0.000000000 )            ( +0.000000000 )

         + ( -0.000000000 ) . sin13t + ( +0.000000000 ) . cos13t
           ( -0.000000000 )            ( -0.000000000 )

         + ( -0.000000000 ) . sin14t + ( +0.000000000 ) . cos14t
           ( -0.000000000 )            ( -0.000000000 )

         + ( +0.000000000 ) . sin15t + ( -0.000000000 ) . cos15t
           ( +0.000000000 )            ( +0.000000000 )
```

Taboau	m	multipliers	
64	+1.741387843e +1	+4.393609058e -1	+0.000000000e +0 i
		+7.907148011e -3	+0.000000000e +0 i
128	+1.741360362e +1	+4.393008902e -1	+0.000000000e +0 i
		+7.907118419e -3	+0.000000000e +0 i
256	+1.741063136e +1	+4.393600892e -1	+0.000000000e +0 i
		+7.907116633e -3	+0.000000000e +0 i

p	r
16	+3.405189350e-11
32	+3.405189350e-11
64	+3.405189350e-11

REFERENCES

[1] VAN DOOREN, R., *Numerical computation of forced oscillations in coupled Duffing equations,* Numer. Math. 20, 300-311, 1973.

[2] HILHORST, D. & J.G. VERWER, *A numerical study of periodicity properties of a Volterra-Lotka system,* Mathematisch Centrum, Amsterdam, to appear.

[3] URABE, M., *Galerkin's procedure for nonlinear periodic systems,* Arch. Rat. Mech. Anal. 20, 120-152, 1965.

[4] URABE, M. & A. REITER, *Numerical computation of nonlinear forced oscillations by Galerkin's procedure,* J. Math. Anal. Appl. 14, 107-140, 1966.

[5] URABE M., *Numerical investigation of subharmonic solutions to Duffing's equation,* Publ. RIMS, Kyoto Univ., Vol. 5, 79-112, 1969.