Object-Oriented Mathematical Programming and Symbolic/Numeric Interface

Larry Lambe*
and
Richard Luczak

June 13, 1993

To appear in the Proc. 3^{rd} International Conf. on Expert Systems in Numerical Computing

Abstract

The Axiom language is based on the notions of "categories", "domains" and "packages". These concepts are used to build an interface between symbolic and numeric calculations. In particular, an interface to the NAG Fortran Library and Axiom's algebra and graphics facilities is presented. Some examples of numerical calculations in a symbolic computational environment are also included using the finite element method. While the examples are elementary, we believe that they point to very powerful methods for combining numeric and symbolic computational techniques.

1. Introduction.

Many problems of interest to engineers and scientists involve the use of numeric libraries. It could be helpful and convenient if the output of numerical routines could be processed directly in a symbolic computational environment. In addition, it could be convenient or revealing if systems could be prepared for numeric processing in a symbolic computational environment.

*First Author partially supported by NSF Grant No. CCR-9207241. He wishes also to thank the Mathematics Department, University of Stockholm for its hospitality during the completion of this paper.

Axiom (formerly called Scratchpad) is a language based on the notions of "domains" and "categories". Using the Axiom System, a very high level of consistent mathematical representation can be quickly obtained. The language supports object oriented paradigms, polymorphism and has a compiler. While the compiler provides a way to produce efficient procedures, there is still a desire to take advantage of the work done by others (e.g., the NAG numerical libraries, personal libraries, etc.).

The main concepts (categories, domains, and packages) of the Axiom language will very briefly be described and it will be shown how one can interface with external programs in Axiom using simple methods. The NAG numeric library and graphics facilities will be used in our examples. Once done, the interface becomes invisible and convenient to the user. This interface will be used to develop a finite element method (FEM) for 2^{nd} order ordinary differential equations (ODE's).

2. Object-Oriented Mathematical Programming.

The Axiom system is capable of supporting a very high level of mathematical programming. For example, if polynomials in several variables over a commutative ring R did not already exist, then one could easily obtain them (except for cosmetic issues explained below). To see this, first, recall that a monoid is a mathematical structure consisting of a set M with an associative operation $* : M \times M \to M$ and an identity element $1 \in M$. If R is any ring, the monoid ring of M over R consists of all finite linear combinations of the form $\sum r_i m_i$ where $r_i \in R$ and $m_i \in M$. It has operations of addition and scalar multiplication (this much gives us the free module of M over R) defined by adding the coefficients of like terms in two formal expressions and multiplying each coefficient of a formal expression by a ring element respectively. It also has a multiplication given by extending the one from M bilinearly, i.e., $(\sum r_i m_i) * (\sum s_i n_i) = \sum \sum r_i s_j m_i * n_j$.

```
-- Except for cosmetics, here are polynomials in one
-- variable. The result is itself a commutative ring.

pol := MonoidRing(R,NNI)

-- Since pol is a ring, we may iterate!  Here are
-- polynomials in two variables.
```

```
--

pol2 := MonoidRing(pol,NNI)
```

What we have given above is valid Axiom code. The lines beginning with `--` are comment lines and we have assumed that the ring `R` has already been defined (some typical rings that already exist in Axiom are `Integer`, and `FiniteField(p,n)`, where `p` is some prime and `n` is a non-negative integer, etc). The domain `NNI` is the monoid of non-negative integers with addition and `0`.

In fact, the free module "constructor" `FreeModule(R,M)` exists in Axiom so even if the constructor `MonoidRing` above did not exist, it would not be difficult to define it.

As indicated, there would be "cosmetic problems" in working with polynomials as defined above (even though it would be mathematically correct to do so). This is because we are used to writing $3t^2 + 2t - 7$ and not $3\ 2 + 2\ 1 - 7\ 0$ for polynomials. However, if we keep in mind that, in an expression of the form $r_1\ n_1 + \ldots + r_k\ n_k$, the terms always come in pairs with the coefficient of the "polynomial" first and the exponent second, then such expressions do indeed add, scalar multiply, and multiply exactly as ordinary polynomials.

Of course, it would not be very convenient if one had to deal with ordinary mathematical domains in some non-standard, foreign way even if it is very quick, convenient and mathematically correct to do so. The Axiom system has a clever solution to this problem, viz., a very rich domain called `OutputForm`. We will say a little more about this domain below.

All functions in Axiom have "signatures". Here we take the word signature to simply mean the specification of the name, domain and range of a function. For example, the function $f(x)$ which takes integers to integers has signature $f : \mathbb{Z} \rightarrow \mathbb{Z}$. The multiplication in some group G has signature $* : G \times G \rightarrow G$; however in Axiom, we would write this latter signature as `*:(G,G) -> G`

We think of an Axiom category as simply a collection of signatures (but more formally it might also have attributes for the given signatures as well as default definitions).

We think of an Axiom domain as an implementation of the signatures from some category. We think of the name of a domain (or sometimes the name along with its arguments) as giving a "domain constructor", for example, the constructor `MonoidRing` (or `MonoidRing(R,M)`) from

above. It is convenient and proper to think of an element in some domain `D` as having type `D`. Similiarly, if `D` is the implementation of the signatures from some category `C`, we think of `D` as having type `C`. Thus, `17` has type `Integer` and `Integer` has type `Ring`.

An Axiom package is simply a domain which contains no signature with a reference to itself in any signature, i.e., it is a collection of (n-ary) functions which operate to and from domains which already exist. See **[Ref. 1], [Ref. 2], [Ref. 3]** for more information about categories, domains, and packages in Axiom.

2.1. OutputForm

The domain `OutputForm` is used by the Axiom system to format expressions. Every Axiom domain `D` must have a signature of the form (possibly inherited from another domain)

```
coerce :  $ -> OutputForm
```

so that expressions from `D` can be displayed. A step in this process is the formation of "output forms" from the underlying data type of `D`.

2.2. Axiom As A Scratchpad

In the work below, we use piecewise polynomial functions for some linearly independant functions Ψ_i mentioned in Section 4 below. For example, take an interval $[a, b]$, partition it into subintervals $[x_i, x_{i+1}]$ where $x_i = a + \frac{b-a}{n-1}(i-1)$, for $i = 1 \ldots n$. Consider the unique piecewise linear or piecewise quadratic function $\Psi_i(x_j) = Q_2^{(i)}$ which satisfies $Q_2^{(i)} = \delta_{i,j}$, i.e., is zero on all nodes x_j for $j \neq i$ and is 1 on x_i. We will not consider other "shape functions" in this paper, but will in a sequel.

In our manipluations, we need a formula for the quadratic shape function mentioned above. The formula is well known, but can easily be found by using the Lagrange approximation formula

$$G(x) = \sum_{j=1}^{n} (\prod_{\substack{i=1 \\ (i \neq j)}}^{n} \frac{x - x_i}{x_j - x_i}) y_j$$

where $x_1, \ldots, x_n$ and $y_1, \ldots y_n$ are arbitrarily chosen points. The function G is a polynomial of degree $n-1$ and, in fact, it is the unique polynomial

of degree $n-1$ which takes the values y_i at x_i. We have implemented the function `lagrange` with signature

```
lagrange :  (List FPI, List FPI) -> FPI
```

where `FPI` is the domain `Fraction Polynomial Integer`. By this we mean that `lagrange` is a function of two variables xval, yval whose values are lists of elements from `FPI` and which returns a rational function (in an indeterminant `x`) with coefficients from `FPI`. The reason for the signature for `lagrange` above is that we want to use this function to actually get Axiom to calculate the formula for the quadratic element mentioned above. We present this to point to a potentially sophisticated use of symbolic manipulation through a simple example. More complicated functions and whole programs have been produced using the paradigm as we will indicate. The implementation of `lagrange` in Axiom is straightforward.

Consider the following Axiom output

```
FPI  := Fraction Polynomial Integer;
PoI  := Polynomial Integer;

ithPt : (PoI, PoI, PoI, PoI) -> FPI;
   ++ ithPt(a,b,n,i) returns the ith point x_i in a uniform
   ++ partition a = x_1 < ... < x_n = b of [a,b]

ithPt(a,b,n,i) ==
   a + (b - a)/(n - 1) * (i - 1)

++
++ Q2(a,b,n,i) = lagrange(lx,ly)
++
++ where lx = [ithPt(a,b,n,i-1), ithPt(a,b,n,i),
++            ithPt(a,b,n,i+1)]
++ and   ly = [0, 1, 0]
++

lx := [ithPt(a,b,n,i-1), ithPt(a,b,n,i), ithPt(a,b,n,i+1)];
ly := [0, 1, 0];

Q2 := lagrange(lx,ly)
```

```
   (9)

          2           2
       (- n  + 2n - 1)x
     +
           2
       (2a n  + ((2b - 2a)i - 2b - 2a)n + (- 2b + 2a)i
     +
              2 2                2                   2
       2b)x - a n  + ((- 2a b + 2a )i + 2a b)n + (- b
     +
              2  2      2                     2
       2a b - a )i  + (2b  - 2a b)i - 2a b + a
  /
      2           2
     b  - 2a b + a
```

```
++ we can make the output look nicer by factoring the
++ numerator and the denominator, isolating the
++ coefficients of the powers of x and factoring those
++ coefficients
++
n := numerator(Q2) :: PoI;
d := denominator(Q2) :: PoI;
listCo := [factor coefficient(n,x,2-i) for i in 0..2]
```

```
   (12)
             2
   [- (n - 1) , 2(n - 1)(a n + (b - a)i - b),
    - (a n + (b - a)i - 2b + a)(a n + (b - a)i - a)]

fd := factor d
                2
   (13)  (b - a)
```

In fact, the resulting expression can be turned into an Axiom procedure or even into procedures in other languages using the techniques in [**Ref.** 4].

3. Symbolic/Numeric Interface.

Using the mathematical programming concepts described above and some more Axiom facilities described below, an Axiom package which implements an interface with the NAG Fortran Library can be constructed.

3.1. An Axiom Package

We have constructed an Axiom package with signature

```
nagLibraryFiniteIntegrate :  (EXPR SF, L L SF,SF) -> L SF
```

where `SF` denotes the domain `SmallFloat` (all double precision floating point numbers), `EXPR SF` denotes the domain `Expression SmallFloat` (all expressions with `SmallFloat` coefficients), `L SF` denotes the domain `List SmallFloat` (all lists of `SmallFloats`), and `L L SF` denotes all lists of lists of `SmallFloats`. Thus, `nagLibraryFiniteIntegrate` is a function of three variables, `ex`, `listIntervals`, and `tol` which returns a list of double precision numbers. The expression `ex` represents some function $f(x)$ to be integrated, `listIntervals` is a list of two element lists $[a_i, b_i]$ representing intervals over which `ex` is to be integrated and `tol` is a tolerance for the precision. The value returned is the list

$$\Big[\int_{a_1}^{b_1} f(x)dx, \ldots, \int_{a_n}^{b_n} f(x)dx\Big] \tag{3.1.1}$$

Each of the integrals in the above list is calculated numerically using the NAG Fortran Library routine D01AKF in double precision **[Ref. 5]**. The routine D01AKF is a high order adaptive integrator, specially suited to oscillating, non-singular integrands.

The actual package also has signature

```
nagLibraryInfiniteIntegrate :  (EXPR SF, I, SF, SF) -> SF
```

where `nagLibraryInfiniteIntegrate(ex,a,err)` returns the definite integral `of` ex over the infinite interval $[a, \infty)$. This function uses the NAG Fortran Library routine D01AMF.

3.2. Implementation

The package `nagLibraryFiniteIntegrate` is a prototype for using Axiom in connection with other programs. There is a sophisticated Axiom

NAG Fortran Library interface that will soon be available in release 2.0 of Axiom, **[Ref. 6]** but we present our method since it is quite transparent, it is not really restricted to any language, and it could be implemented by any ordinary Axiom user without much trouble using only available tools.

The NAG Fortran Library interface we present needs only three simple facilities in Axiom.

- It is possible to execute system commands from within Axiom. This is done by typing `systemCommand(cmd)`, where `cmd` is some command string (e.g., "ls -l").
- It is possible to create, read and write to files containing elements from some domain.
- It is possible to automatically translate Axiom expressions to Fortran and write them to a file.

Using these tools,

```
nagLibraryFiniteIntegrate(ex, listIntervals, tol)
```

works as follows. First, two files to hold the translated expression and the interval data are opened. Then an external program (which was written in C) that constructs, compiles and executes a Fortran program using the NAG Fortran Library to do the integrations is executed. The results of that program are written to an ordinary ascii file. Next, the Axiom procedure opens the result file as a file of `SmallFloat` and it reads the answers into an Axiom list. It then simply returns that list as its value.

The Axiom and C language source files for all of this as well as the polynomial interpolation package mentioned above and below are available from the authors. In the following Axiom log, we assume that these packages have been loaded.

3.3. Example

Let f be the function

$$f(x) = \frac{\sin\left(7\ \cos\left(2\ x^2 + 4\right) + 1\right)}{\sqrt[3]{x^4 + 1}}$$

where $x \in [0, 1]$. This is an oscillating, non-singular function. Let $a_1 = \ldots = a_n = 0$ and $b_i = \frac{i-1}{n-1}$ for $i = 1, \ldots, n$. Under these assumptions,

(3.1.1) takes the form

$$\Big[\int_0^{b_1} f(x)dx, \ldots, \int_0^{b_n} f(x)dx\Big]$$

The integrands in the above formula were evaluated using the Axiom command

```
nagLibraryFiniteIntegrate
```

Below is the output from an interactive Axiom session.

```
(1) ->ex:=sin(7*cos(2*x**2 +4) +1)/(x**4+1)**(1/3)

                    2
        sin(7cos(2x  + 4) + 1)
   (1)  ----------------------
                +------+
               3| 4
               \|x  + 1
                                    Type: Expression Integer

(2) ->listIntervals := [ [0.0, k/10] for k in 0..10];

(3) ->tol := 0.000001;

(4) ->yval := nagLibraryFiniteIntegrate(ex,listIntervals,_
              tol)

   using double precision in the NAG library call ...

** __main   === End of Compilation 1 ===
** fst   === End of Compilation 2 ===
1501-510  Compilation successful for file 518478.f.

   (4)
   [0.0, 0.038776082000000003, 0.056710366000000005,
    0.030133671000000001, - 0.050921549000000003,
    - 0.13388517999999999, - 0.11137420000000001,
    - 0.034221694000000004, - 0.081859037999999995,
    - 0.12621963999999999, - 0.066157009000000003]
                                       Type: List SmallFloat
```

Using a sequence of such lists of numbers we used the Axiom system to produce a sequence of polynomial approximations to the integral. The approximations utilized Lagrange's polynomial interpolation formula (2.1).

Let's form the Lagrange's interpolating polynomial of order 10. The output from interactive Axiom session is given below.

```
(5) ->xval := [xx.2 for xx in listIntervals]

   (5)  [0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
                                                Type: List Float
(6) ->lagr := lagrange(xval,yval)

   (6)
                          10                        9
     - 484.70799988974505x   - 246.10142884710297x
   +
                         8                         7
     6001.1724148481353x   - 13111.337085235136x
   +
                         6                         5
     13249.759952853236x  - 7402.1151778965959x
   +
                         4                         3
     2399.3018225495221x  - 447.59894818807999x
   +
                         2
     42.750684703885327x  - 1.1903919072301827x
                                Type: Polynomial SmallFloat
```

In a similar way we can create Lagrange's interpolating polynomial of order 15. The graphical comparison of the interpolants is created by using the graphics tools available in Axiom. The plot is presented in Fig. 1.

4. A Finite Element Method for Second Order Ordinary Linear Differential Equations.

We briefly recall some of the basic ideas behind the finite element method (FEM) in a very special case in this section. Our basic reference is **[Ref. 7]**. We will only deal with real valued functions of a real variable and, for the sake of exposition, we will not deal with things like singularities here.

4.1 The method

Let $\mathcal{F}(a,b)$ denote a set of functions $f : [a,b] \to \mathbb{R}$ where $a, b \in \mathbb{R}$ and f is "sufficiently" differentiable for the disscussion below.

Suppose that $D : \mathcal{F}(a,b) \to \mathcal{F}(a,b)$ is some operator, e.g., $D(u) = a_2 u'' + a_1 u' + a_0 u$, for fixed $a_i \in \mathcal{F}(a,b)$. We will use a standard inner product on $\mathcal{F}(a,b)$ given by $< f,\ g > = \int_a^b fg$.

We are interested in accurately approximating the solution to the equation $D(u) = r$ for a fixed $r \in \mathcal{F}(a,b)$.

Roughly speaking, we can think of the finite element method as consisting of the following:

(4.1.1) Try to find a function u such that for all functions $v \in \mathcal{F}(a,b)$ we have $< v,\ D(u) > = < v,\ r >$.

(4.1.2) Use some transformation to rewrite the left hand side $< v,\ D(u) >$ in a more "convenient form" if possible (see below).

(4.1.3) Choose some "convenient" set of linearly independent functions $\mathcal{B} = \{\Psi_i\}_{i=1}^n \subset \mathcal{F}(a,b)$ and try to solve the transformed equation of (4.1.1) for u in the linear span of $\mathcal{B}$.

In practice, a finite set of equations is obtained by allowing v above to range only through the linear span of $\mathcal{B}$. Thus, one attempts to solve the system

$$(4.1.4) \qquad < \Psi_i,\ D(u) > = < \Psi_i,\ r >, \quad i = 1 \ldots, n$$

for $u = \sum_{j=1}^n u_j \Psi_j$.

4.2 Second Order Linear ODE's

We will consider the general 2^{nd} order linear boundry value/initial value problem

$$Du = f \tag{4.2.1}$$
$$\alpha_0 u(a) + \alpha_1 u'(a) = \gamma_1, \quad \beta_0 u(b) + \beta_1 u'(b) = \gamma_2$$

where $\alpha_i, \beta_i, \gamma_j \in \mathbb{R}$, $Du = a_2 u'' + a_1 u' + a_0 u$ and $a_2 \neq 0$ on the interval $[a, b]$.

Following **Ref. 7**, we observe that (4.2.1) is equivalent to

$$(pu')' + qu = r \tag{4.2.2}$$
$$\alpha_0 u(a) + \alpha_1 u'(a) = \gamma_1, \quad \beta_0 u(b) + \beta_1 u'(b) = \gamma_2$$

where

$$p(x) = exp \int_a^x \frac{a_1}{a_2}, \; q = \frac{a_0 p}{a_2}, \; r = \frac{fp}{a_2} \tag{4.2.3}$$

We will consider the method outlined above in (4.1) for (4.2.2). Indeed, by straightforward manipulations including "integration by parts" from elementary calculus, we obtain

$$[\Psi_i(b)p(b)u'(b) - \Psi_i(a)p(a)u'(a)] + \sum (S_{i,j} - R_{i,j})u_j = B_i \tag{4.2.4}$$

directly from (4.1.4) where

$$R_{i,j} = < \Psi_i', \; p\Psi_j' >, \quad S_{i,j} = < \Psi_i, \; q\Psi_j >, \quad \text{and} \quad B_i = < \Psi_i, \; r > .$$

Case i) $\alpha_1 \neq 0, \beta_1 \neq 0$ The boundry conditions become

$$u'(a) = (\gamma_1 - \alpha_0 u(a))/\alpha_1, \quad u'(b) = (\gamma_2 - \beta_0 u(b))/\beta_1$$

We assume that we have an approximate solution of the form $u = \sum_{i=1}^n u_i \Psi_i$ where $\Psi_1(a) = 1$, $\Psi_n(b) = 1$ and $\Psi_i(a) = 0$ if $i \neq 1$, $\Psi_i(b) = 0$ if $i \neq n$.

The system (4.2.4) can be written in the form

$$\begin{aligned}
(S_{1,1} - R_{1,1} + \frac{\alpha_0 p(a)}{\alpha_1})u_1 + \sum_{j=2}^{n} (S_{1,j} - R_{1,j})u_j &= B_1 + \frac{\gamma_1 p(a)}{\alpha_1} \\
\sum_{j=1}^{n} (S_{i,j} - R_{i,j})u_j &= B_i, \; 2 \leq i \leq n-1 \\
\sum_{j=1}^{n-1} (S_{n,j} - R_{n,j})u_j + (S_{n,n} - R_{n,n} - \frac{\beta_0 p(b)}{\beta_1})u_n &= B_n - \frac{\gamma_2 p(b)}{\beta_1}
\end{aligned}$$

Using a similiar approach we can write appropriate systems for the remaining three cases:
Case ii) $\alpha_1 \neq 0, \beta_1 = 0$

In this case, $u_n = u(b) = \gamma_2/\beta_0$ is known, thus (4.2.4) is needed only for $1 \leq i \leq n-1$ and it becomes

$$(S_{1,1} - R_{1,1} + \frac{\alpha_0 p(a)}{\alpha_1})u_1 + \sum_{j=2}^{n-1}(S_{1,j} - R_{1,j})u_j =$$

$$B_1 + \frac{\gamma_1 p(a)}{\alpha_1} - (S_{1,n} - R_{1,n})u_n$$

$$\sum_{j=1}^{n-1}(S_{i,j} - R_{i,j})u_j = B_i - (S_{i,n} - R_{i,n})u_n,\ 2 \leq i \leq n-1$$

Case iii) $\alpha_1 = 0, \beta_1 \neq 0$

In this case, $u_1 = u(a) = \gamma_1/\alpha_0$ is known, thus (4.2.4) is needed only for $2 \leq i \leq n$ and it becomes

$$\sum_{j=2}^{n}(S_{i,j} - R_{i,j})u_j = B_i - (S_{i,1} - R_{i,1})u_1,\ 2 \leq i \leq n-1$$

$$\sum_{j=2}^{n-1}(S_{n,j} - R_{n,j})u_j + (S_{n,n} - R_{n,n} - \frac{\beta_0 p(b)}{\beta_1})u_n =$$

$$B_n - \frac{\gamma_2 p(b)}{\beta_1} - (S_{n,1} - R_{n,1})u_1$$

Case iv) $\alpha_1 = 0, \beta_1 = 0$

In this case, $u_1 = u(a) = \gamma_1/\alpha_0$ and $u_n = u(b) = \gamma_2/\beta_0$ are known, thus (4.2.4) is needed only for $2 \leq i \leq n-1$ and it becomes

$$\sum_{j=2}^{n-1}(S_{i,j} - R_{i,j})u_j = B_i - (S_{i,1} - R_{i,1})u_1 - (S_{i,n} - R_{i,n})u_n,\ 2 \leq i \leq n-1$$

4.3 Example

Let us consider the problem descibed in Section 4.2, Case iv), i.e.

$$(pu')' + qu = r \tag{4.3.1}$$

with boundary conditions of the form

$$u(a) = u_a, \quad u(b) = u_b$$

We will note that each such problem can be transformed to one with homogeneous boundary conditions. The transformation has the form

$$u(x) = w(x) + c_1 x + c_2 \tag{4.3.2}$$

where $w(x)$ is a new unknown function, $c_1, c_2 \in \mathbb{R}$ are given by

$$c_1 = \frac{u_a - u_b}{a - b}, \quad c_2 = \frac{a u_b - b u_a}{a - b}$$

The transformation (4.3.2) applied to (4.3.1) produces $(pw')' + qw = h$ where

$$h(x) = r(x) - c_1 p' - (c_1 x + c_2) q \tag{4.3.3}$$

As an example consider the problem

$$u'' + u = -sin(x) \tag{4.3.4}$$

with boundary conditions of the form

$$u(a) = u_a, \quad u(b) = u_b$$

It can be shown that the problem (4.3.4) has the solution

$$u(x) = d_1 cos(x) + d_2 sin(x) + (2x cos(x) - sin(x))/4$$

where

$$\begin{aligned} d_1 &= \frac{(2u_a - a cos(a)) sin(b) + (b cos(b) - 2u_b) sin(a)}{2(sin(b)cos(a) - sin(a)cos(b))} \\ d_2 &= \frac{(sin(b) + 4u_b) cos(a) + (2(a-b)cos(a) - 2u_a - sin(a)) cos(b)}{4(sin(b)cos(a) - sin(a)cos(b))} \end{aligned}$$

To solve this problem in Axiom the method described in Section 4.1 will be applied. Let Ω_n be a uniform mesh over $[a, b]$,

$$\Omega_n = \{x : x = a + ih, i = 1, ..., n, \; h = \frac{b-a}{n-1}\}$$

Consider the function $\phi(s)$ of the form

$$\phi(s) = \begin{cases} 1 - |s|, & |s| < 1 \\ 0, & |s| \geq 1 \end{cases}$$

Then each function $\Psi_i \in \mathcal{B},\ i = 1, ..., n$ can be written in the form:

$$\Psi_i(x) = \phi(\frac{x - a}{h} - i),\ i = 1, ..., n$$

Functions $\Psi_i, i = 1, ..., n$ are piecewise linear and linearly independent. They form the space $\mathcal{B} = \{\Psi_i\}_{i=1}^n$. The following data were used: $a = 0.1, b = 1.3, u_a = 0.12, u_b = 0.05$. The plot is presented in Fig.2.

Kent State University, Kent, OH 44242 and
NAG, Downers Grove, IL 60515

NAG, Downers Grove, IL 60515

Fig.1 Lagrange's interpolating polynomial of order n; n=10 - solid line with dots, n=15 - solid line.

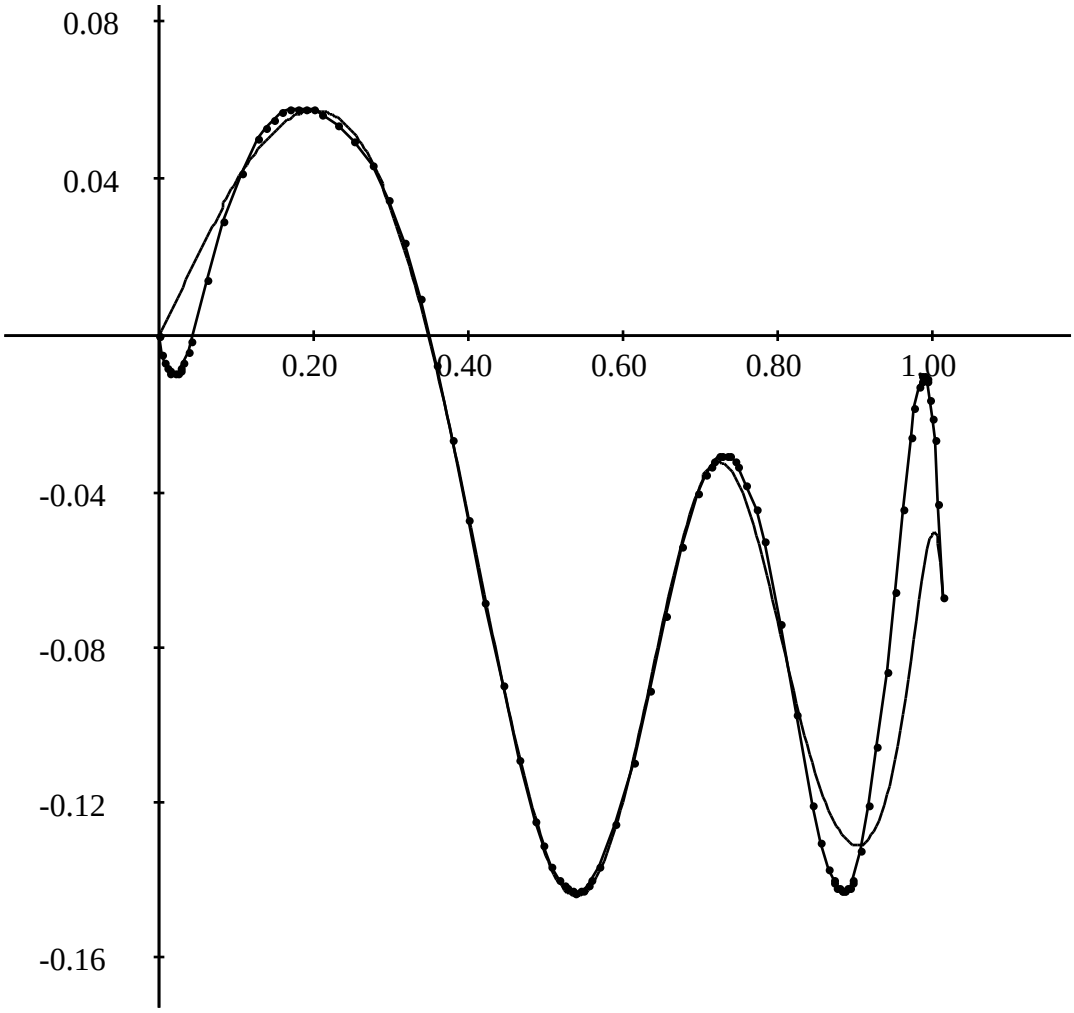
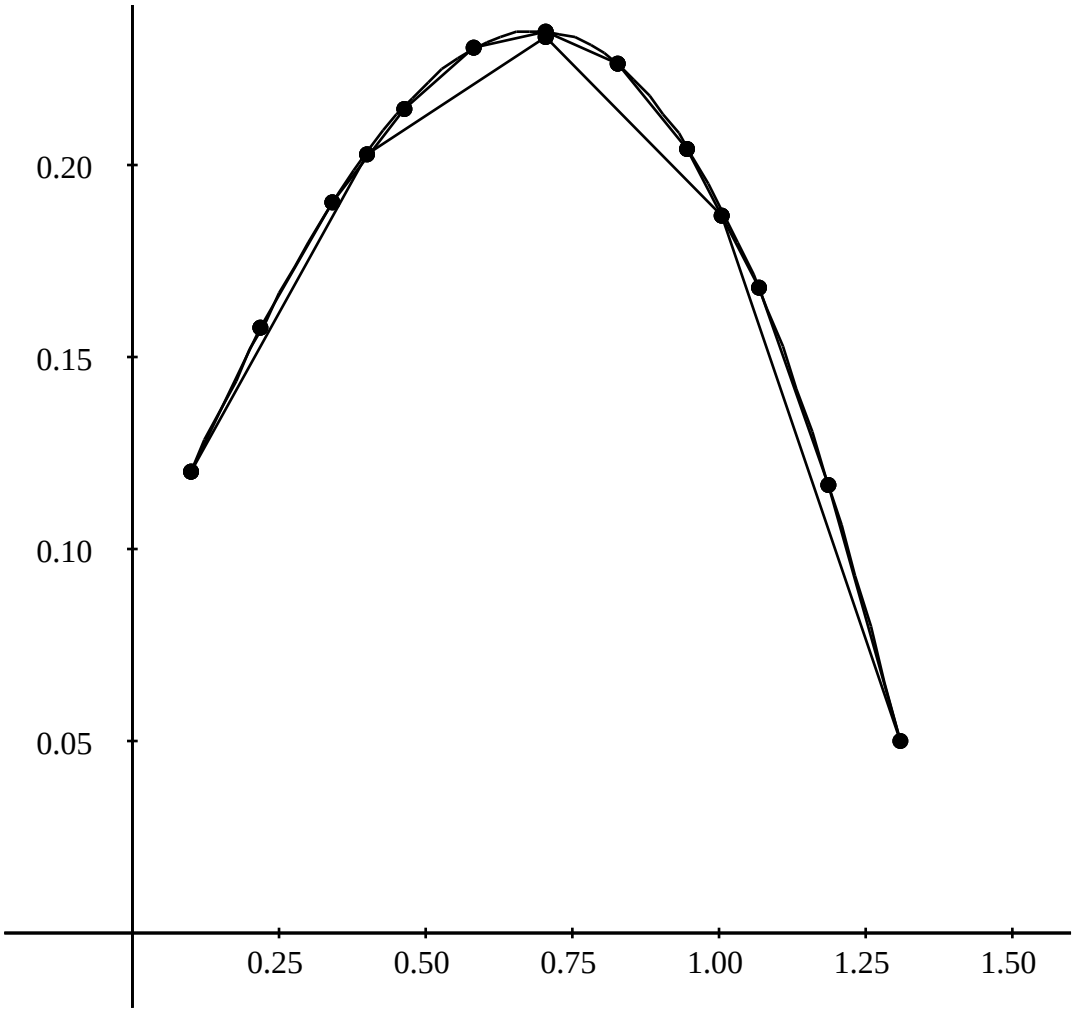


Fig. 2 Finite element method applied to problem (4.3.4); exact solution - solid line, n=5 and n=11 - piecewise linear elements.



References

[1] Lambe, Larry, Scratchpad II as a tool for mathematical research, Jon Barwise's column in Notices of the Amer. Math. Soc., February, 1989.

[2] Lambe, Larry, Next Generation Computer Algebra Systems, AXIOM and the Scratchpad Concept: Applications to Research in Algebra, 21^{st} Nordic Congress of Mathematicians, Luleå, Sweden, Summer, 1992, (to appear).

[3] Jenks, Richard and Sutor, Robert, **Axiom**, Springer-Verlag, (1992).

[4] Lambe, Larry, On Using Axiom to Generate Code, (preprint), 1993.

[5] The NAG Fortran Library Manual, Mark 15, Volume 1, **D01AKF - NAG Fortran Library Routine Document**.

[6] Dewar, Mike, Manipulating FORTRAN code in AXIOM and the AXIOM-NAG Link, Univ. Bath, to appear in: **The Proceedings of the Second Workshop on Symbolic and Numeric Computation**, Helsinki, (1993).

[7] Lewis, P. E. and Ward, J. P., **The Finite Element Method, Principles and Applications**, Addison Wesley, Wokingham, England, Reading, MA, (1991).