4

Analysis of Example Truss by a CAS

TABLE OF CONTENTS

Page

§4.1. **Computer Algebra Systems** 4–3
- §4.1.1. Why Mathematica? 4–3
- §4.1.2. How to Get It? 4–3
- §4.1.3. Programming Style and Prerequisites 4–4
- §4.1.4. Class Demo Scripts 4–4

§4.2. **The Element Stiffness Module** 4–4
- §4.2.1. Module Description 4–6
- §4.2.2. Programming Remarks 4–6
- §4.2.3. Case Sensitivity 4–6
- §4.2.4. Testing the Member Stiffness Module 4–7

§4.3. **Merging a Member into the Master Stiffness** 4–8

§4.4. **Assembling the Master Stiffness** 4–9

§4.5. **Modifying the Master System** 4–9

§4.6. **Recovering Internal Forces** 4–11

§4.7. **Putting the Pieces Together** 4–12
- §4.7.1. The Driver Program 4–13
- §4.7.2. Is All of This Worthwhile? 4–13

§4. **Notes and Bibliography**. 4–14

§4. **References**. 4–15

§4. **Exercises** 4–16

§4.1. Computer Algebra Systems

Computer algebra systems, known by the acronym CAS, are programs designed to perform symbolic and numeric manipulations following the rules of mathematics.[1] The development of such programs began in the mid 1960s. The first comprehensive system — the "granddaddy" of them all, called *Macsyma* (an acronym for Project **Mac Sy**mbolic **Ma**nipulator) — was developed using the programming language Lisp at MIT's famous Artificial Intelligence Laboratory over the period 1967 to 1980.

The number and quality of symbolic-manipulation programs has expanded dramatically since the availability of graphical workstations and personal computers has encouraged interactive and experimental programming. As of this writing the leading general-purpose contenders are *Maple* and *Mathematica*.[2] In addition there are a dozen or so more specialized programs, some of which are available free or at very reasonable cost. See **Notes and Bibliography** at the end of the Chapter.

§4.1.1. Why Mathematica?

In the present book *Mathematica* will be used for Chapters and Exercises that develop symbolic and numerical computation for matrix structural analysis and FEM implementations. *Mathematica* is a commercial product developed by Wolfram Research, web site: `http://www.wolfram.com`. The version used to construct the code fragments presented in this Chapter is 4.1, which was commercially released in 2001. (The latest version is 5.2, released 2005) The main advantages of *Mathematica* for technical computing are:

1. Availability on a wide range of platforms that range from PCs and Macs through Unix workstations.
2. Up-to-date user interface. On all machines *Mathematica* offers a graphics user interface called the Notebook front-end. This is mandatory for serious work. It provides advanced result typesetting.
3. A powerful programming language.
4. Good documentation and abundance of application books at all levels.

One common disadvantage of CAS, and *Mathematica* is no exception, is computational inefficiency in numerical calculations compared with a low-level implementation in, for instance, C or Fortran. The relative penalty can reach several orders of magnitude. For instructional use, however, the penalty is acceptable when compared to *human efficiency*. This means the ability to get FEM programs up and running in very short time, with capabilities for symbolic manipulation and graphics as a bonus.

§4.1.2. How to Get It?

Registered students may purchase the student version for about $150 at any campus bookstore. You will need to show proof you are a bona-fide student at the register.[3] If you are not on campus (e.g. a CAETE student) you may purchase it directly at the vendor's web site `http://www.wolfram.com`. Again proof of registration must be provided.

The student version is a bargain, since the standard personal license costs over $1K. Unlike other commercial software products, you get the full thing; no capabilities are emasculated. But terms are

[1] Some vendors call that kind of activity "doing mathematics by computer." It is more appropriate to regard such programs as enabling tools that help humans with complicated and error-prone manipulations. Mettle *and* metal. As of now, only humans can do mathematics.

[2] Another commonly used program for engineering computations: *Matlab*, does only numerical computations although a [poorly done] interface to *Maple* can be purchased as a toolbox.

[3] You should check for discounts when new versions come out; unsold previous-version copies may go for as little as $50.

very strict: once installed on your laptop or desktop, it cannot be transferred since the license is forever keyed to the disk identification.

You can also get a 1-year license from CU's Information Technology Services (ITS) for $119, renewable yearly. Check the list of site-licensable software at `http://www.colorado.edu/ITS/TPSiteLic`. If you plan to keep *Mathematica* for more than a year, however, the student version is a better deal.

§4.1.3. Programming Style and Prerequisites

The following material assumes that you are a moderately experienced user of *Mathematica*, or are willing to learn to be one. See **Notes and Bibliography** for a brief discussion of tutorial and reference materials.

Practice with the program until you reach the level of writing functions, modules and scripts with relative ease. With the Notebook interface and a good primer it takes only a few hours.

When approaching that level you may notice that functions in *Mathematica* display many aspects similar to C.[4] You can exploit this similarity if you are proficient in that language. But *Mathematica* functions do have some unique aspects, such as matching arguments by pattern, and the fact that internal variables are global unless otherwise made local.[5]

Although function arguments can be modified, in practice this should be avoided because it may be difficult to trace side effects. The programming style enforced here outlaws output arguments and a function can only return its name. But since the name can be a list of arbitrary objects the restriction is not serious.[6]

Our objective is to develop a symbolic program written in *Mathematica* that solves the example plane truss as well as some symbolic versions thereof. The program will rely heavily on the development and use of *functions* implemented using the `Module` construct of *Mathematica*. Thus the style will be one of procedural programming.[7] The program will not be particularly modular (in the computer science sense) because *Mathematica* is not suitable for that programming style.[8]

The code presented in Sections 4.2–4.7 uses a few language constructs that may be deemed as advanced, and these are briefly noted in the text so that appropriate reference to the *Mathematica* reference manual can be made.

§4.1.4. Class Demo Scripts

The cell scripts shown in Figures 4.1 and 4.2 will be used to illustrate the organization of a Notebook file and the "look and feel" of some basic *Mathematica* commands. These scripts will be demonstrated in class from a laptop.

---

[4] Simple functions can be implemented in `Mathematica` directly, for instance `DotProduct[x_,y_]:=x.y;` more complicated functions are handled by the `Module` construct. These things are called *rules* by computer scientists.

[5] In *Mathematica* everything is a function, including programming constructs. Example: in C `for` is a loop-opening keyword; in *Mathematica* `For` is a function that runs a loop according to its arguments.

[6] Such restrictions on arguments and function returns are closer in spirit to `C` than `Fortran` although you can of course modify C-function arguments using pointers — exceedingly dangerous but often unavoidable.

[7] The name `Module` should not be taken too seriously: it is far away from the concept of modules in Ada, Modula, Oberon or Fortran 90. But such precise levels of interface control are rarely needed in symbolic languages.

[8] And indeed none of the CAS packages in popular use is designed for strong modularity because of historical and interactivity constraints.

Integration example

```
f[x_,α_,β_]:=(1+β*x^2)/(1+α*x+x^2);
F=Integrate[f[x,-1,2],{x,0,5}];
F=Simplify[F];
Print[F]; Print[N[F]];
F=NIntegrate[f[x,-1,2],{x,0,5}];
Print["F=",F//InputForm];
```

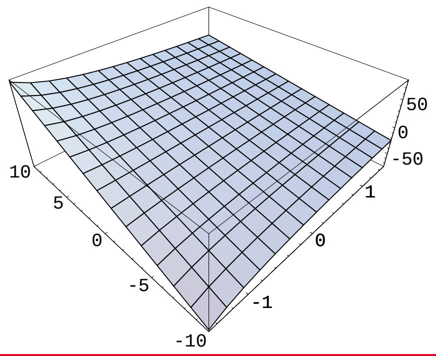
```
10 + Log[21]
13.0445
F=13.044522437723455
```

FIGURE 4.1. Example cell for class demo.

```
Fa=Integrate[f[z,a,b],{z,0,5}]; Fa=Simplify[Fa]; Print["Fa=",Fa];
Plot3D[Fa,{a,-1.5,1.5},{b,-10,10},ViewPoint->{-1,-1,1}];
Fa=FullSimplify[Fa]; (* very slow but you get *)  Print["Fa=",Fa];
```

$$
\begin{aligned}
\mathrm{Fa}=\frac{1}{2\sqrt{4-a^2}}\Bigg(&10\sqrt{4-a^2}\,b-a\sqrt{4-a^2}\,b\,\mathrm{Log}[26+5a]-i\,(2+(-2+a^2)\,b)\,\mathrm{Log}\Big[1-\frac{i\,a}{\sqrt{4-a^2}}\Big]+\\
&2\,i\,\mathrm{Log}\Big[1+\frac{i\,a}{\sqrt{4-a^2}}\Big]-2\,i\,b\,\mathrm{Log}\Big[1+\frac{i\,a}{\sqrt{4-a^2}}\Big]+i\,a^2\,b\,\mathrm{Log}\Big[1+\frac{i\,a}{\sqrt{4-a^2}}\Big]+\\
&2\,i\,\mathrm{Log}\Big[\frac{-10\,i-i\,a+\sqrt{4-a^2}}{\sqrt{4-a^2}}\Big]-2\,i\,b\,\mathrm{Log}\Big[\frac{-10\,i-i\,a+\sqrt{4-a^2}}{\sqrt{4-a^2}}\Big]+i\,a^2\,b\,\mathrm{Log}\Big[\frac{-10\,i-i\,a+\sqrt{4-a^2}}{\sqrt{4-a^2}}\Big]-\\
&2\,i\,\mathrm{Log}\Big[\frac{10\,i+i\,a+\sqrt{4-a^2}}{\sqrt{4-a^2}}\Big]+2\,i\,b\,\mathrm{Log}\Big[\frac{10\,i+i\,a+\sqrt{4-a^2}}{\sqrt{4-a^2}}\Big]-i\,a^2\,b\,\mathrm{Log}\Big[\frac{10\,i+i\,a+\sqrt{4-a^2}}{\sqrt{4-a^2}}\Big]\Bigg)
\end{aligned}
$$

Result after Simplify[..]



Result after FullSimplify[..]

$$
\mathrm{Fa}=5\,b-\frac{1}{2}\,a\,b\,\mathrm{Log}[26+5a]-\frac{i\,(2+(-2+a^2)\,b)\left(\mathrm{Log}\Big[1-\frac{i\,a}{\sqrt{4-a^2}}\Big]-\mathrm{Log}\Big[1+\frac{i\,a}{\sqrt{4-a^2}}\Big]-\mathrm{Log}\Big[1-\frac{i\,(10+a)}{\sqrt{4-a^2}}\Big]+\mathrm{Log}\Big[1+\frac{i\,(10+a)}{\sqrt{4-a^2}}\Big]\right)}{2\sqrt{4-a^2}}
$$

FIGURE 4.2. Another example cell for class demo.

§4.2. The Element Stiffness Module

As our first FEM code segment, the top box of Figure 4.3 shows a module that evaluates and returns the 4 × 4 stiffness matrix of a plane truss member (two-node bar) in global coordinates. The text in that box of that figure is supposed to be placed on a Notebook cell. Executing the cell, by clicking on it and hitting an appropriate key (`<Enter>` on a Mac), gives the output shown in the bottom box. The contents of the figure is described in further detail below.

§4.2.1. Module Description

The stiffness module is called `ElemStiff2DTwoNodeBar`. Such descriptive names are permitted by the language. This reduces the need for detailed comments.

The module takes two arguments:

`{ { x1 , y1 } , { y1 , y2 } }`	A two-level list[9] containing the $\{x, y\}$ coordinates of the bar end nodes labelled as 1 and 2.[10]
`{ Em , A }`	A one-level list containing the bar elastic modulus, E and the member cross section area, A. See §4.2.3 as to why name `E` cannot be used.

The use of the underscore after argument item names in the declaration of the `Module` is a requirement for pattern-matching in *Mathematica*. If, as recommended, you have learned functions and modules this aspect should not come as a surprise.

The module name returns the 4 × 4 member stiffness matrix internally called `Ke`. The logic that leads to the formation of that matrix is straightforward and need not be explained in detail. Note, however, the elegant direct declaration of the matrix `Ke` as a level-two list, which eliminates the fiddling around with array indices typical of standard programming languages. The format in fact closely matches the mathematical expression (2.18).

§4.2.2. Programming Remarks

The function in Figure 4.3 uses several intermediate variables with short names: `dx`, `dy`, `s`, `c` and `L`. It is strongly advisable to make these symbols *local* to avoid potential names clashes somewhere else.[11] In the `Module[ ...]` construct this is done by listing those names in a list immediately after the opening bracket. Local variables may be *initialized* when they are constants or simple functions of the argument items; for example on entry to the module `dx=x2-x1` initializes variable `dx` to be the difference of x node coordinates, namely $\Delta x = x_2 - x_1$.

The use of the `Return` statement fulfills the same purpose as in C or Fortran 90. *Mathematica* guides and textbooks advise against the use of that and other C-like constructs. The writer strongly disagrees: the `Return` statement makes clear what the `Module` gives back to its invoker and is self-documenting.

[9] A level-one list is a sequence of items enclosed in curly braces. For example: { x1,y1 } is a list of two items. A level-two list is a list of level-one lists. An important example of a level-two list is a matrix.

[10] These are called the *local node numbers*, and replace the i, j of previous Chapters. This is a common FEM programming practice.

[11] The “global by default” choice is the worst one, but we must live with the rules of the language.

```
ElemStiff2DTwoNodeBar[{{x1_,y1_},{x2_,y2_}},{Em_,A_}] :=
  Module[{c,s,dx=x2-x1,dy=y2-y1,L,Ke},
    L=Sqrt[dx^2+dy^2]; c=dx/L; s=dy/L;
    Ke=(Em*A/L)* {{ c^2, c*s,-c^2,-c*s},
                  { c*s, s^2,-s*c,-s^2},
                  {-c^2,-s*c, c^2, s*c},
                  {-s*c,-s^2, s*c, s^2}};
    Return[Ke]
 ];
Ke= ElemStiff2DTwoNodeBar[{{0,0},{10,10}},{100,2*Sqrt[2]}];
Print["Numerical elem stiff matrix:"]; Print[Ke//MatrixForm];
Ke= ElemStiff2DTwoNodeBar[{{0,0},{L,L}},{Em,A}];
Ke=Simplify[Ke,L>0];
Print["Symbolic elem stiff matrix:"]; Print[Ke//MatrixForm];
```

```
Numerical elem stiff matrix:
```

$$\begin{pmatrix} 10 & 10 & -10 & -10 \\ 10 & 10 & -10 & -10 \\ -10 & -10 & 10 & 10 \\ -10 & -10 & 10 & 10 \end{pmatrix}$$

```
Symbolic elem stiff matrix:
```

$$\begin{pmatrix} \frac{A\,Em}{2\sqrt{2}\,L} & \frac{A\,Em}{2\sqrt{2}\,L} & -\frac{A\,Em}{2\sqrt{2}\,L} & -\frac{A\,Em}{2\sqrt{2}\,L} \\ \frac{A\,Em}{2\sqrt{2}\,L} & \frac{A\,Em}{2\sqrt{2}\,L} & -\frac{A\,Em}{2\sqrt{2}\,L} & -\frac{A\,Em}{2\sqrt{2}\,L} \\ -\frac{A\,Em}{2\sqrt{2}\,L} & -\frac{A\,Em}{2\sqrt{2}\,L} & \frac{A\,Em}{2\sqrt{2}\,L} & \frac{A\,Em}{2\sqrt{2}\,L} \\ -\frac{A\,Em}{2\sqrt{2}\,L} & -\frac{A\,Em}{2\sqrt{2}\,L} & \frac{A\,Em}{2\sqrt{2}\,L} & \frac{A\,Em}{2\sqrt{2}\,L} \end{pmatrix}$$

FIGURE 4.3. Module `ElemStiff2DTwoNodeBar` to form the element stiffness of a 2D 2-node truss element in global coordinates. Test statements (in blue) and test output.

§4.2.3. Case Sensitivity

Mathematica, like most recent computer languages, is case sensitive so that for instance `E` is not the same as `e`. This is fine. But the language designer decided that names of system-defined objects such as built-in functions and constants must begin with a capital letter. Consequently the liberal use of names beginning with a capital letter may run into clashes. For example you cannot use `E` because of its built-in meaning as the base of natural logarithms.[12]

In the code fragments presented throughout this book, identifiers beginning with upper case are used for objects such as stiffness matrices, modulus of elasticity, and cross section area, following established usage in Mechanics. When there is danger of clashing with a protected system symbol, additional lower case letters are used. For example, `Em` is used for the elastic modulus instead of `E` because the latter is a reserved symbol.

§4.2.4. Testing the Member Stiffness Module

Following the definition of `ElemStiff2DTwoNodeBar` in Figure 4.3 there are several statements that constitute the *module test program* that call the module and print the returned results. Two cases are tested. First, the stiffness of member (3) of the example truss, using all-numerical values. Next, some of the input arguments for the same member are given symbolic names so they stand for variables; for

[12] In retrospect this appears to have been a highly questionable decision. System defined names should have been identified by a reserved prefix or postfix to avoid surprises, as done in *Macsyma* or *Maple*. *Mathematica* issues a warning message, however, if an attempt to redefine a "protected symbol" is made.

```
MergeElemIntoMasterStiff[Ke_,eftab_,Kin_]:=Module[
{i,j,ii,jj,K=Kin},
   For [i=1, i<=4, i++, ii=eftab[[i]];
      For [j=i, j<=4, j++, jj=eftab[[j]];
           K[[jj,ii]]=K[[ii,jj]]+=Ke[[i,j]]
      ]
   ]; Return[K]
];
K=Table[0,{6},{6}];
Print["Initialized master stiffness matrix:"];
Print[K//MatrixForm]
Ke=ElemStiff2DTwoNodeBar[{{0,0},{10,10}},{100,2*Sqrt[2]}];
Print["Member stiffness matrix:"]; Print[Ke//MatrixForm];
K=MergeElemIntoMasterStiff[Ke,{1,2,5,6},K];
Print["Master stiffness after member merge:"];
Print[K//MatrixForm];
```

```
Initialized master stiffness matrix:
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
Member stiffness matrix:
```

$$\begin{pmatrix} 10 & 10 & -10 & -10 \\ 10 & 10 & -10 & -10 \\ -10 & -10 & 10 & 10 \\ -10 & -10 & 10 & 10 \end{pmatrix}$$

```
Master stiffness after member merge:
```

$$\begin{pmatrix} 10 & 10 & 0 & 0 & -10 & -10 \\ 10 & 10 & 0 & 0 & -10 & -10 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -10 & -10 & 0 & 0 & 10 & 10 \\ -10 & -10 & 0 & 0 & 10 & 10 \end{pmatrix}$$

FIGURE 4.4. Module `MergeElemIntoMasterStiff` to merge a 4×4 bar element stiffness into the master stiffness matrix. Test statements (in blue) and test output.

example the elastic module is given as `Em` instead of 100 as in the foregoing test. The print output of the test is shown in the lower portion of Figure 4.3.

The first test returns the member stiffness matrix (2.21) as may be expected. The second test returns a symbolic form in which three symbols appear: the coordinates of end node 2, which is taken to be located at `{ L,L }` instead of {10, 10}, `A`, which is the cross-section area and `Em`, which is the elastic modulus. Note that the returning matrix `Ke` is subject to a `Simplify` step before printing it, which is the subject of an Exercise. The ability to carry along variables is of course a fundamental capability of any CAS and the main reason for which such programs are used.

§4.3. Merging a Member into the Master Stiffness

The next fragment of *Mathematica* code, listed in Figure 4.4, is used in the assembly step of the DSM. Module `MergeElemIntoMasterStiff` receives the 4×4 element stiffness matrix formed by `FormElemStiff2DNodeBar` and “merges” it into the master stiffness matrix. The module takes three arguments:

`Ke`	The 4×4 member stiffness matrix to be merged. This is a level-two list.
`eftab`	The column of the Element Freedom Table, defined in §3.4.1, appropriate to the member

```
AssembleMasterStiffOfExampleTruss[]:=
Module[{Ke,K=Table[0,{6},{6}]},
    Ke=ElemStiff2DTwoNodeBar[{{0,0},{10,0}},{100,1}];
    K= MergeElemIntoMasterStiff[Ke,{1,2,3,4},K];
    Ke=ElemStiff2DTwoNodeBar[{{10,0},{10,10}},{100,1/2}];
    K= MergeElemIntoMasterStiff[Ke,{3,4,5,6},K];
    Ke=ElemStiff2DTwoNodeBar[{{0,0},{10,10}},{100,2*Sqrt[2]}];
    K= MergeElemIntoMasterStiff[Ke,{1,2,5,6},K];
    Return[K]
  ];
K=AssembleMasterStiffOfExampleTruss[];
Print["Master stiffness of example truss:"]; Print[K//MatrixForm];
```

```
Master stiffness of example truss:
```

$$\begin{pmatrix} 20 & 10 & -10 & 0 & -10 & -10 \\ 10 & 10 & 0 & 0 & -10 & -10 \\ -10 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & -5 \\ -10 & -10 & 0 & 0 & 10 & 10 \\ -10 & -10 & 0 & -5 & 10 & 15 \end{pmatrix}$$

FIGURE 4.5. Module `AssembleMasterStiffOfExampleTruss` that forms the 6×6 master stiffness matrix of the example truss. Test statements (in blue) and test output.

being merged; cf. (3.22). Recall that the EFT lists the global equation numbers for the four member degrees of freedom. This is a level-one list consisting of 4 integers.

`Kinp` The incoming 6×6 master stiffness matrix. This is a level-two list.

`MergeElemIntoMasterStiff` returns, as module name, the updated master stiffness matrix internally called `K` with the member stiffness merged in. Thus we encounter here a novelty: an input-output argument. Because a formal argument cannot be modified, the situation is handled by copying the incoming `Kin` into `K` on entry. It is the copy which is updated and returned via the `Return` statement. The implementation has a strong C flavor with two nested `For` loops. Because the iterators are very simple, nested `Do` loops could have been used as well.

The statements after the module provide a simple test. Before the first call to this function, the master stiffness matrix must be initialized to a zero 6×6 array. This is done in the first test statement using the `Table` function. The test member stiffness matrix is that of member (3) of the example truss, and is obtained by calling `ElemStiff2DTwoNodeBar`. The EFT is `{1,2,5,6}` since element freedoms 1,2,3,4 map into global freedoms 1,2,5,6. Running the test statements yields the listing given in Figure 4.4. The result is as expected.

§4.4. Assembling the Master Stiffness

The module `AssembleMasterStiffOfExampleTruss`, listed in the top box of Figure 4.5, makes use of the foregoing two modules: `ElemStiff2DTwoNodeBar` and `MergeElemIntoMasterStiff`, to form the master stiffness matrix of the example truss. The initialization of the stiffness matrix array in `K` to zero is done by the `Table` function of `Mathematica`, which is handy for initializing lists. The remaining statements are self explanatory. The module is similar in style to argumentless Fortran or C functions. It takes no arguments. All the example truss data is “wired in.”

The output from the test program in is shown in the lower box of Figure 4.5. The output stiffness matches that in Equation (3.12), as can be expected if all code fragments used so far work correctly.

```
ModifiedMasterStiffForDBC[pdof_,K_] := Module[
  {i,j,k,nk=Length[K],np=Length[pdof],Kmod=K},
      For [k=1,k<=np,k++, i=pdof[[k]];
         For [j=1,j<=nk,j++, Kmod[[i,j]]=Kmod[[j,i]]=0];
           Kmod[[i,i]]=1];
   Return[Kmod]
];
ModifiedMasterForcesForDBC[pdof_,f_] := Module[
  {i,k,np=Length[pdof],fmod=f},
      For [k=1,k<=np,k++, i=pdof[[k]]; fmod[[i]]=0];
   Return[fmod]
];
K=Array[Kij,{6,6}]; Print["Assembled master stiffness:"];
Print[K//MatrixForm];
K=ModifiedMasterStiffForDBC[{1,2,4},K];
Print["Master stiffness modified for displacement B.C.:"];
Print[K//MatrixForm];
f=Array[fi,{6}]; Print["Force vector:"]; Print[f];
f=ModifiedMasterForcesForDBC[{1,2,4},f];
Print["Force vector modified for displacement B.C.:"]; Print[f];
```

```
Assembled master stiffness:
⎛Kij[1, 1]  Kij[1, 2]  Kij[1, 3]  Kij[1, 4]  Kij[1, 5]  Kij[1, 6]⎞
⎜Kij[2, 1]  Kij[2, 2]  Kij[2, 3]  Kij[2, 4]  Kij[2, 5]  Kij[2, 6]⎟
⎜Kij[3, 1]  Kij[3, 2]  Kij[3, 3]  Kij[3, 4]  Kij[3, 5]  Kij[3, 6]⎟
⎜Kij[4, 1]  Kij[4, 2]  Kij[4, 3]  Kij[4, 4]  Kij[4, 5]  Kij[4, 6]⎟
⎜Kij[5, 1]  Kij[5, 2]  Kij[5, 3]  Kij[5, 4]  Kij[5, 5]  Kij[5, 6]⎟
⎝Kij[6, 1]  Kij[6, 2]  Kij[6, 3]  Kij[6, 4]  Kij[6, 5]  Kij[6, 6]⎠

Master stiffness modified for displacement B.C.:
⎛1  0     0      0     0          0     ⎞
⎜0  1     0      0     0          0     ⎟
⎜0  0  Kij[3, 3]  0  Kij[3, 5]  Kij[3, 6]⎟
⎜0  0     0      1     0          0     ⎟
⎜0  0  Kij[5, 3]  0  Kij[5, 5]  Kij[5, 6]⎟
⎝0  0  Kij[6, 3]  0  Kij[6, 5]  Kij[6, 6]⎠
Force vector:
{fi[1], fi[2], fi[3], fi[4], fi[5], fi[6]}
Force vector modified for displacement B.C.:
{0, 0, fi[3], 0, fi[5], fi[6]}
```

FIGURE 4.6. Modules `ModifiedMasterStiffForDBC` and `ModifiedMasterForceForDBC` that modify the master stiffness matrix and force vector of a truss to impose displacement BCs. Test statements (in blue) and test output.

§4.5. Modifying the Master System

Following the assembly process the master stiffness equations $\mathbf{K}\mathbf{u} = \mathbf{f}$ must be modified to account for single-freedom displacement boundary conditions. This is done through the computer-oriented equation modification process outlined in §3.5.2.

Module `ModifiedMasterStiffForDBC` carries out this process for the master stiffness matrix $\mathbf{K}$, whereas `ModifiedMasterForcesForDBC` does this for the nodal force vector $\mathbf{f}$. These two modules are listed in the top box of Figure 4.6, along with test statements. The logic of both functions, but especially that of `ModifiedMasterForcesForBC`, is considerably simplified by assuming that *all prescribed displacements are zero*, that is, the BCs are homogeneous. The more general case of nonzero prescribed values is treated in Part III of the book.

Function `ModifiedMasterStiffnessForDBC` has two arguments:

`pdof`	A list of the prescribed degrees of freedom identified by their global number. For the example truss this list contains three entries: {1, 2, 4}.

```
IntForce2DTwoNodeBar[{{x1_,y1_},{x2_,y2_}},{Em_,A_},eftab_,u_]:=
  Module[ {c,s,dx=x2-x1,dy=y2-y1,L,ix,iy,jx,jy,ubar,e},
    L=Sqrt[dx^2+dy^2]; c=dx/L; s=dy/L; {ix,iy,jx,jy}=eftab;
    ubar={c*u[[ix]]+s*u[[iy]],-s*u[[ix]]+c*u[[iy]],
          c*u[[jx]]+s*u[[jy]],-s*u[[jx]]+c*u[[jy]]};
    e=(ubar[[3]]-ubar[[1]])/L; Return[Em*A*e]
];
p =IntForce2DTwoNodeBar[{{0,0},{10,10}},{100,2*Sqrt[2]},
         {1,2,5,6},{0,0,0,0,0.4,-0.2}];
Print["Member int force (numerical):"]; Print[N[p]];
p =IntForce2DTwoNodeBar[{{0,0},{L,L}},{Em,A},
         {1,2,5,6},{0,0,0,0,ux3,uy3}];
Print["Member int force (symbolic):"]; Print[Simplify[p]];
```

```
Member int force (numerical):
2.82843
Member int force (symbolic):
A Em (ux3 + uy3)
     2 L
```

FIGURE 4.7. Module `IntForce2DTwoNodeBar` for computing the internal force in a bar element. Test statements (in blue) and test output.

`K` The master stiffness matrix **K** produced by the assembly process.

The function clears appropriate rows and columns of **K**, places ones on the diagonal, and returns the modified **K** as function value. The only slightly fancy thing in this module is the use of the *Mathematica* function `Length` to extract the number of prescribed displacement components: `Length[pdof]` here will return the value 3, which is the length of the list `pdof`. Similarly `nk=Length[K]` assigns 6 to `nk`, which is the order of matrix **K**. Although for the example truss these values are known *a priori*, the use of `Length` serves to illustrate a technique that is heavily used in more general code.

Module `ModifiedMasterForcesForDBC` has similar structure and logic and need not be described in detail. It is important to note, however, that for homogeneous BCs the modules are independent of each other and may be called in any order. On the other hand, if there were nonzero prescribed displacements present the force modification must be done *before* the stiffness modification. This is because stiffness coefficients that are cleared in the latter are needed for modifying the force vector.

The test statements are purposedly chosen to illustrate another feature of *Mathematica*: the use of the `Array` function to generate subscripted symbolic arrays of one and two dimensions. The test output is shown in the bottom box of Figure 4.6, which should be self explanatory. The force vector and its modified form are printed as row vectors to save space.

§4.6. Recovering Internal Forces

Mathematica provides built-in matrix operations for solving a linear system of equations and multiplying matrices by vectors. Thus we do not need to write application functions for the solution of the modified stiffness equations and for the recovery of nodal forces. Consequently, the last application functions we need are those for internal force recovery.

Function `IntForce2DTwoNodeBar` listed in the top box of Figure 4.7 computes the internal force in an individual bar element. It is somewhat similar in argument sequence and logic to `ElemStiff2DTwoNodeBar` of Figure 4.3. The first two arguments are identical. Argument `eftab` provides the Element Freedom Table array for the element. The last argument, `u`, is the vector of computed node displacements.

The logic of `IntForce2DTwoNodeBar` is straightforward and follows the method outlined in §3.2.1.

```
IntForcesOfExampleTruss[u_]:= Module[{f=Table[0,{3}]},
  f[[1]]=IntForce2DTwoNodeBar[{{0,0},{10,0}},{100,1},{1,2,3,4},u];
  f[[2]]=IntForce2DTwoNodeBar[{{10,0},{10,10}},{100,1/2},{3,4,5,6},u];
  f[[3]]=IntForce2DTwoNodeBar[{{0,0},{10,10}},{100,2*Sqrt[2]},
         {1,2,5,6},u];
  Return[f]
];
f=IntForcesOfExampleTruss[{0,0,0,0,0.4,-0.2}];
Print["Internal member forces in example truss:"];Print[N[f]];
```

```
Internal member forces in example truss:
{0., -1., 2.82843}
```

FIGURE 4.8. Module `IntForceOfExampleTruss` that computes internal forces in the 3 members of the example truss. Test statements (in blue) and test output.

Member joint displacements $\bar{\mathbf{u}}^{(e)}$ in local coordinates $\{\bar{x}, \bar{y}\}$ are recovered in array `ubar`, then the longitudinal strain $e = (\bar{u}_{xj} - \bar{u}_{xi})/L$ and the internal (axial) force $p = EAe$ is returned as function value. As coded the function contains redundant operations because entries 2 and 4 of `ubar` (that is, components $\bar{u}_{yi}$ and $\bar{u}_{yj}$) are not actually needed to get p, but were kept to illustrate the general backtransformation of global to local displacements.

Running this function with the test statements shown after the module produces the output shown in the bottom box of Figure 4.7. The first test is for member (3) of the example truss using the actual nodal displacements (3.17). It also illustrates the use of the *Mathematica* built in function `N` to produce output in floating-point form. The second test does a symbolic calculation in which several argument values are fed in variable form.

The top box of Figure 4.8 lists a higher-level function, `IntForceOfExampleTruss`, which has a single argument: `u`. This is the complete 6-vector of joint displacements $\mathbf{u}$. This function calls `IntForce2DTwoNodeBar` three times, once for each member of the example truss, and returns the three member internal forces thus computed as a 3-component list.

The test statements listed after `IntForcesOfExampleTruss` feed the actual node displacements (3.24) to `IntForcesOfExampleTruss`. Running the functions with the test statements produces the output shown in the bottom box of Figure 4.8. The internal forces are $p^{(1)} = 0$, $p^{(2)} = -1$ and $p^{(3)} = 2\sqrt{2} = 2.82843$.

§4.7. Putting the Pieces Together

After all this development and testing effort documented in Figures 4.3 through 4.8 we are ready to make use of all these bits and pieces of code to analyze the example plane truss. This is actually done with the logic shown in Figure 4.9. This *driver program* uses the seven previously described modules

```
ElemStiff2DTwoNodeBar
MergeElemIntoMasterStiff
AssembleMasterStiffOfExampleTruss
ModifiedMasterStiffForDBC
ModifiedMasterForcesForDBC
IntForce2DTwoNodeTruss
IntForcesOfExampleTruss
```
(4.1)

These functions must have been defined (”compiled”) at the time the driver programs described below

```
f={0,0,0,0,2,1};
K=AssembleMasterStiffOfExampleTruss[];
Kmod=ModifiedMasterStiffForDBC[{1,2,4},K];
fmod=ModifiedMasterForcesForDBC[{1,2,4},f];
u=Simplify[Inverse[Kmod].fmod];
Print["Computed nodal displacements:"]; Print[u];
f=Simplify[K.u];
Print["External node forces including reactions:"]; Print[f];
p=Simplify[IntForcesOfExampleTruss[u]];
Print["Internal member forces:"]; Print[p];
```

```
Computed nodal displacements:
{0, 0, 0, 0, 2/5, -1/5}
External node forces including reactions:
{-2, -2, 0, 1, 2, 1}
Internal member forces:
{0, -1, 2√2}
```

FIGURE 4.9. Driver program for numerical analysis of example truss and its output.

are run. A simple way to making sure that all of them are defined is to put all these functions in the same Notebook file and to mark them as *initialization cells*. These cells may be executed by picking up Kernel → Initialize → Execute Initialization. (An even simpler procedure would to group them all in one cell, but that would make placing separate test statements difficult.)

For a hierarchical version of (4.1), see the last CAETE slide.

§4.7.1. The Driver Program

The program listed in the top box of Figure 4.9 first assembles the master stiffness matrix through `AssembleMasterStiffOfExampleTruss`. Next, it applies the displacement boundary conditions through `ModifiedMasterStiffForDBC` and `ModifiedMasterForcesForDBC`. Note that the modified stiffness matrix is placed into `Kmod` rather than `K` to save the original form of the master stiffness for the reaction force recovery later. The complete displacement vector is obtained by the matrix calculation

$$\texttt{u=Inverse[Kmod].fmod} \tag{4.2}$$

which takes advantage of two built-in *Mathematica* functions. `Inverse` returns the inverse of its matrix argument[13] The dot operator signifies matrix multiply (here, matrix-vector multiply.) The enclosing `Simplify` function in Figure 4.9 is asked to simplify the expression of vector `u` in case of symbolic calculations; it is actually redundant if all computations are numerical.

The remaining statements recover the node vector including reactions via the matrix-vector multiply `f = K.u` (recall that `K` contains the unmodified master stiffness matrix) and the member internal forces `p` through `IntForcesOfExampleTruss`. The program prints `u`, `f` and `p` as row vectors to save space.

Running the program of the top box of Figure 4.9 produces the output shown in the bottom box of that figure. The results confirm the hand calculations of Chapter 3.

§4.7.2. Is All of This Worthwhile?

At this point you may wonder whether all of this work is worth the trouble. After all, a hand calculation (typically helped by a programable calculator) would be quicker in terms of flow time. Typing and

[13] This is a highly inefficient way to solve $\mathbf{K}\mathbf{u} = \mathbf{f}$ if this system becomes large. It is done here to keep simplicity.

```
f={0,0,0,0,fx3,fy3};
K=AssembleMasterStiffOfExampleTruss[];
Kmod=ModifiedMasterStiffForDBC[{1,2,4},K];
fmod=ModifiedMasterForcesForDBC[{1,2,4},f];
u=Simplify[Inverse[Kmod].fmod];
Print["Computed nodal displacements:"]; Print[u];
f=Simplify[K.u];
Print["External node forces including reactions:"]; Print[f];
p=Simplify[IntForcesOfExampleTruss[u]];
Print["Internal member forces:"]; Print[p];
```

```
Computed nodal displacements:
              1                    1
{0, 0, 0, 0, ──(3 fx3 - 2 fy3),  ─(-fx3 + fy3)}
             10                    5
External node forces including reactions:
{-fx3, -fx3, 0, fx3 - fy3, fx3, fy3}
Internal member forces:
{0, -fx3 + fy3, √2 fx3}
```

FIGURE 4.10. Driver program for symbolic analysis of example truss and its output.

debugging the *Mathematica* fragments displayed here took the writer about six hours (although about two thirds of this was spent in editing and getting the fragment listings into the Chapter.) For larger problems, however, *Mathematica* would certainly beat hand-plus-calculator computations, the cross-over typically appearing for 10 to 20 equations. For up to about 500 equations and using floating-point arithmetic, *Mathematica* gives answers within minutes on a fast PC or Mac with sufficient memory but eventually runs out of steam at about 1000 equations. For a range of 1000 to about 50000 equations, *Matlab*, using built-in sparse solvers, would be the best compromise between human and computer flow time. Beyond 50000 equations a program in a low-level language, such as C or Fortran, would be most efficient in terms of computer time.[14]

One distinct advantage of computer algebra systems emerges when you need to *parametrize* a small problem by leaving one or more problem quantities as variables. For example suppose that the applied forces on node 3 are to be left as f_{x3} and f_{y3}. You replace the last two components of array p as shown in the top box of Figure 4.10, execute the cell and shortly get the symbolic answer shown in the bottom box of that figure. This is the answer to an infinite number of numerical problems. Although one may try to undertake such studies by hand, the likelyhood of errors grows rapidly with the complexity of the system. Symbolic manipulation systems can amplify human abilities in this regard, as long as the algebra "does not explode" because of combinatorial complexity. Examples of such nontrivial calculations will appear throughout the following Chapters.

Remark 4.1. The "combinatorial explosion" danger of symbolic computations should be always kept in mind. For example, the numerical inversion of a $N \times N$ matrix is a $O(N^3)$ process, whereas symbolic inversion goes as $O(N!)$. For $N = 48$ the floating-point numerical inverse will be typically done in a fraction of a second. But the symbolic adjoint will have $48! = 12413915592536072670862289047373375038521486354677760000000000$ terms, or $O(10^{61})$. There may be enough electrons in this Universe to store that, but barely ...

[14] The current record for FEM structural applications is about 100 million equations, done on a massively parallel supercomputer (ASCI Red at SNL). Fluid mechanics problems with over 500 million equations have been solved.

Notes and Bibliography

As noted in §4.1.2 the hefty *Mathematica Book* [191] is a reference manual. Since the contents are available online (click on **Help** in topbar) as part of purchase of the full system,[15] buying the printed book is optional.[16]

There is a nice tutorial available by Glynn and Gray [78], list: \$35, dated 1999. (Theodore Gray invented the Notebook front-end that appeared in version 2.2.) It is also available on CDROM from MathWare, Ltd, P. O. Box 3025, Urbana, IL 61208, e-mail: `info@mathware.com`. The CDROM is a hyperlinked version of the book that can be installed on the same directory as *Mathematica*. More up to date and comprehensive is the recent appeared *Mathematica Navigator* in two volumes [145,146]; list: \$69.95 but used copies are discounted, down to about \$20.

Beyond these, there are many books at all levels that expound on the use of *Mathematica* for various applications ranging from pure mathematics to physics and engineering. A web search (September 2003) on `www3.addall.com` hit 150 book titles containing *Mathematica*, compared to 111 for *Maple* and 148 for *Matlab*. A `google` search (August 2005) hits 3,820,000 pages containing *Mathematica*, but here *Matlab* wins with 4,130,000.

Wolfram Research hosts the MathWorld web site at `http://mathworld.wolfram.com`, maintained by Eric W. Weisstein. It is essentially a hyperlinked, on-line version of his *Encyclopædia of Mathematics* [182].

To close the topic of symbolic versus numerical computation, here is a nice summary by A. Grozin, posted on the Usenet:

> "Computer Algebra Systems (CASs) are programs [that] operate with formulas. Mathematica is a powerful CAS (though quite expensive). Other CASs are, e.g., Maple, REDUCE, MuPAD <...>. There are also quite powerful free CASs: Maxima and Axiom. In all of these systems, it is possible to do some numerical calculations (e.g., to evaluate the formula you have derived at some numerical values of all parameters). But it is a very bad idea to do large-scale numerical work in such systems: performance will suffer. In some special cases (e.g., numerical calculations with very high precision, impossible at the double-precision level), you can use Mathematica to do what you need, but there are other, faster ways to do such things.
>
> There are a number of programs to do numerical calculations with usual double-precision numbers. One example is Matlab; there are similar free programs, e.g., Octave, Scilab, R, ... Matlab is very good and fast in doing numerical linear algebra: if you want to solve a system of 100 linear equations whose coefficients are all numbers, use Matlab; if coefficients contain letters (symbolic quantities) and you want the solution as formulas, use Mathematica or some other CAS. Matlab can do a limited amount of formula manipulations using its "symbolic toolkit," which is an interface to a cut-down Maple. It's a pain to use this interface: if you want Maple, just use Maple."

References

Referenced items have been moved to Appendix R.

[15] The student version comes with limited help.

[16] The fifth edition, covering version 5, lists for \$49.95 but older editions are heavily discounted on the web, some under \$2.

Homework Exercises for Chapter 4
Analysis of Example Truss by a CAS

Before doing any of these Exercises, download the *Mathematica* Notebook file `ExampleTruss.nb` from the course web site. (Go to Chapter 5 Index and click on the link). Open this Notebook file using version 4.0 or a later one. The first eight cells contain the modules and test statements listed in the top boxes of Figures 4.3–10. The first six of these are marked as *initialization cells*. Before running driver programs, they should be executed by picking up Kernel → Evaluation → Execute Initialization. Verify that the output of those six cells agrees with that shown in the bottom boxes of Figures 4.3–6. Then execute the driver programs in Cells 7–8 by clicking on each cell and pressing the appropriate key: <Enter> on a Mac, <Shift-Enter> on a Windows PC. Compare the output with that shown in Figures 4.9–10. If the output checks out, you may proceed to the Exercises.

EXERCISE 4.1 [C:10] Explain why the `Simplify` command in the test statements of Figure 4.3 says `L>0`. (One way to figure this out is to just say `Ke=Simplify[Ke]` and look at the output. Related question: why does `Mathematica` refuse to simplify `Sqrt[L^2]` to `L` unless one specifies the sign of `L` in the `Simplify` command?

EXERCISE 4.2 [C:10] Explain the logic of the `For` loops in the merge function `MergeElemIntoMasterStiff` of Figure 4.4. What does the operator `+=` do?

EXERCISE 4.3 [C:10] Explain the reason behind the use of `Length` in the modules of Figure 4.6. Why not simply set `nk` and `np` to 6 and 3, respectively?

EXERCISE 4.4 [C:15] Of the seven modules listed in Figures 4.3 through 4.8, with names collected in (4.2), two can be used only for the example truss, three can be used for any plane truss, and two can be used for other structures analyzed by the DSM. Identify which ones and briefly state the reasons for your classification.

EXERCISE 4.5 [C:20] Modify the modules `AssembleMasterStiffOfExampleTruss`, `IntForcesOfExampleTruss` and the driver program of Figure 4.9 to solve numerically the three-node, two-member truss of Exercise 3.6. Verify that the output reproduces the solution given for that problem. Hint: modify cells but keep a copy of the original Notebook handy in case things go wrong.

EXERCISE 4.6 [C:25] Expand the logic of `ModifiedMasterForcesForDBC` to permit specified nonzero displacements. Specify these in a second argument called `pval`, which contains a list of prescribed values paired with `pdof`.

```
xynode={{0,0},{10,0},{10,10}}; elenod={{1,2},{2,3},{3,1}};
unode={{0,0},{0,0},{2/5,-1/5}}; amp=5;  p={};
For [t=0,t<=1,t=t+1/5,
    For [e=1,e<=Length[elenod],e++, {i,j}=elenod[[e]];
    xyi=xynode[[i]];ui=unode[[i]];xyj=xynode[[j]];uj=unode[[j]];
    p=AppendTo[p,Graphics[Line[{xyi+amp*t*ui,xyj+amp*t*uj}]]];
        ];
    ];
Show[p,Axes->False,AspectRatio->Automatic];
```

FIGURE E4.1. Mystery program for Exercise 4.7.

EXERCISE 4.7 [C:20] Explain what the program of Figure E4.1 does, and the logic behind what it does. (You may want to put it in a cell and execute it.) What modifications would be needed so it can be used for any plane struss?