

Integrated Engineering Development Environment

Oliver Rübenkönig
Zhenyu Liu
Jan G. Korvink

The *IMTEK Mathematica Supplement (IMS)* provides the basis for an Integrated Engineering Development Environment (IEDE). In this context we created an open source linear finite element modeling environment comprising the mechanics, fluidics, and a general differential operators domain. We split the operators from the geometry and shape functions, resulting in symbolic operators, which are well suited for finite element code generation, and fast numerical operators. In this article we show how we use the fast numerical operators for analyzing a microelectromechanical system device. A fast transient solution by means of model order reduction is presented. Finally, we conclude with a harmonic analysis.

■ Introduction

The *IMTEK Mathematica Supplement (IMS)* is a downloadable open source add-on package for *Mathematica* [1]. The supplement provides several hundred functions in about 40 packages. At IMTEK (the Institut für Mikrosystemtechnik) we encounter a variety of engineering tasks. The tasks range from modeling to analysis and culminate in the ultimate engineering goal—good design.

Unfortunately, the design process is often tedious for several reasons. It may be very difficult to extend existing Integrated Engineering Development Environments (IEDEs) due to their being closed source. Exporting some parts of the design process to controllable external tools and applying new algorithms is time-consuming. We try to remedy this shortcoming by providing the open source basis for an IEDE embedded in *Mathematica*.

In order to evaluate this notebook, *IMS* must be installed. Although *IMS* is a growing environment, we try to maintain the functionality between different versions. Sometimes, however, obsolete warning messages may appear. This means that a specific function has been replaced by a newer one. The code, however, should still work correctly.

■ Integrated Engineering Development Environment

An IEDE consists of several pieces that can be smoothly interchanged to reach a design. The ability to model and simulate several different physical domains (e.g., the electrical engineering and mechanical domains) must be combined with the ability to analyze.

This can be accomplished with an adequate data structure, which must be applicable to other—not yet conceived—domains. We will demonstrate this process with a Finite Element Method (FEM) analysis example and later show how this concept is extendable. The finite element code is based on [2, 3, 4].

We start from the observation that engineering systems are constructed by some “basic” components that are related to each other via some connectivity, to be further specified. This connectivity can then be transformed into a system of ordinary differential equations (ODEs) (Figure 1).

Some connectivity \longrightarrow System of ODEs

Figure 1. Common to many engineering applications is the scenario that components are somehow connected, and, by some means, they and their representations in reality have a system of ODEs describing them.

A typical engineering application would be the components of an electrical circuit such as resistors, inductors, capacitances, and sources. These elements are connected to each other via “nodes” (Figure 2). The elements and nodes reside in a graph. The elements are not restricted to circuit elements, but can be elements such as finite elements that the engineer might need.

The system of equations is a system of second-order ODEs. The coefficients are the mass, damping, and stiffness matrices. The load is a vector of a list of multiple load vectors.

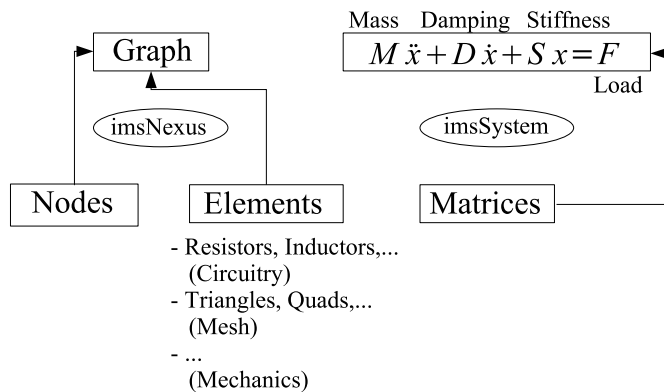


Figure 2. The connectivity in engineering applications can be captured with graphs consisting of nodes and elements, where the different element types come from different application areas. In *IMS* the data structure representing graphs is called *imsNexus*. The system of ODEs is represented by a system of second-order ODEs with matrix coefficients. The underlying data structures are matrices.

In this article we show how we use the graph data structure, apply finite element operators in the elements in the graph, and obtain a system of equations (Figure 3). The finite element operators are taken from the classical scalar partial differential equations (PDEs). The concept, however, is general, and for different engineering domains different operators can assemble a system of equations.

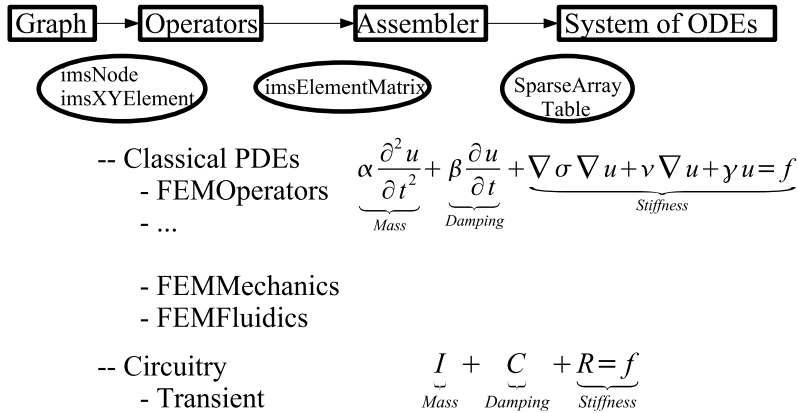


Figure 3. An operator can be applied to each of the elements in a graph. Every operator returns an element matrix (an `imsElementMatrix` data structure). This is then assembled into a global matrix by the assembler routine. The operators can come from a wide range of engineering areas.

■ Heat Anemometer Application Example

Here we model a heat flow anemometer. Figure 4 shows a schematic of the device.

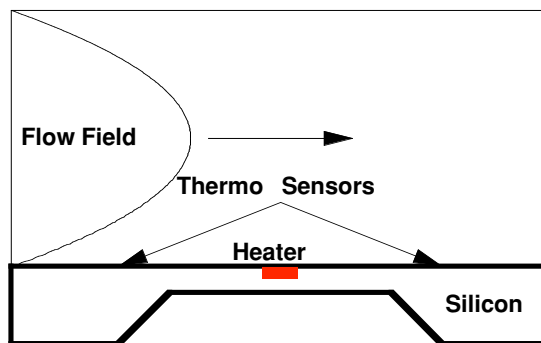


Figure 4. A schematic of a heat flow anemometer.

In a piece of bulk silicon we have embedded a small heater device. To the right and left are two temperature sensors. If the heater is switched on, both sensors will measure the same temperature. Once a flow field is flowing over the silicon device, the temperature distribution will shift and the left sensor will be cooler than the right sensor. From the temperature difference we can calculate the speed of the flow field. This is the principle of an anemometer. More about thermal measurements in fluids can be found in [5].

In the next section we calculate the temperature distribution in the device and its surrounding area.

■ The Equation

In the most general assumption, we wish to model the following equation:

$$\underbrace{\sigma \nabla^2 u}_{\text{Diffusion}} + \underbrace{\gamma \nabla u}_{\text{Convection}} = \underbrace{f}_{\text{Load}} \quad (1)$$

Stiffness Matrix Load Matrix

The heater element will be modeled with a load matrix and the airflow with the convection term. The general heat transfer is modeled with the diffusion term.

■ Loading Predefined Mesh

We begin by loading the *Imtek`Interfaces`EasyMesh`* package.

```
In[86]:= Needs["Imtek`Interfaces`EasyMesh`"]
```

Then we import the *EasyMesh*-generated example mesh.

```
In[87]:= {numberOfNodes, mesherNodes, numberOfElements,
  mesherElements, numberOfSides, sides} = imsReadEasyMesh[
  ToFileName[{ "Imtek", "IMSData", "ApplicationExamples", "FEM"},
  "AnemometerMesh"], imsStartNodeNumbering → 1];
```

□ Creating the Nodes

First we load the following packages.

```
In[88]:= Needs["Imtek`Graph`"]
Needs["Imtek`MeshElementLibrary`"]
```

Then we take the raw nodes and make *IMS* nodes from them. Each of the raw nodes is a list of a unique node identification number (id), the coordinates $\{x, y\}$, and a marker. This marker was inserted by the mesh generator and indicates whether the node belongs to one of the domain segments or not.

We also divide the nodes into boundary nodes and interior nodes. Boundary nodes are the nodes on the simulation domain boundary that specify the bound-

ary conditions. In the mesh input file we have specified that markers 1, 2, and 3 are on the simulation boundary. All other nodes are interior nodes. Either they belong to interior domain segments (and carry the marker 7) or they do not belong to any segment at all (and carry the marker 0).

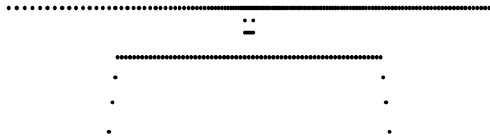
The *IMS* nodes we create also carry a unique identification number (*id*), the coordinates $\{x, y\}$, the marker, a value of the solution (initially set to 0), and the type of the boundary condition.

```
In[90]:= interiorNodes =
  Select[mesherNodes, (#[[4]] == 0 || #[[4]] == 7) &] /.
    {id_, x_, y_, marker_} => imsMakeNode[id, {x, y}, marker];
```

Here we select from the list of all mesh nodes those entries where the marker is 0 or 7.

This displays all the nodes that have a marker of 7.

```
In[91]:= Show[Graphics[Point[#] & /@
  (#[{2, 3}]) & /@ Select[mesherNodes, (#[[4]] == 7) &]],
  AspectRatio -> 0.25]
```



The boundary nodes are those nodes that have a fourth entry not equal to 0 or 7. Additionally we set boundary conditions in the nodes according to the markers for the specific node. The nodes now carry the *id*, the coordinates, the marker, the boundary condition value, and the boundary condition type.

```
In[92]:= boundaryNodes =
  Select[mesherNodes, (#[[4]] != 0 && #[[4]] != 7) &] /.
    {id_, x_, y_, marker_} =>
      Which[
        marker == 1,
          imsMakeNode[id, {x, y}, marker, {{0.}}, "Dirichlet"],
        marker == 2, imsMakeNode[id, {x, y},
          marker, {{0.}}, "Dirichlet"],
        marker == 3, imsMakeNode[id, {x, y},
          marker, {{0.}}, "Dirichlet"]
      ];
```

□ Creating the Elements

Now we set up the elements. Dirichlet boundary conditions are only set up in the nodes. Neumann boundary conditions also need an entry in the elements. Each element consisting of two Neumann boundary nodes will additionally carry the boundary values for the nodes involved.

First we join all nodes and obtain the Neumann boundary node ids.

