I. Characterization of the problem

The author of mathematical software rarely has the luxury of composing a program specifically to perform the task immediately before him, i.e. in a specific context, with specific data, on a specific machine. The cost of creating mathematical software is too high, and the number of individuals capable of producing a particular piece of mathematical software is too small, for such directedness to be acceptable. Instead, mathematical software typically has to be portable, so it can readily be made to run in many different machine environments; it has to be modular, so it can be integrated into many other programs; and it must be adaptable, so alternate data structures, algorithms, and other design decisions can be used. An item of mathematical software is thus not a single program, but a family of programs. (Actually in most programming languages the item is often not a program as such, but rather a package of smaller units).

In other words, mathematical software today requires the ability to produce multiple realizations of the same conceptual program. Portability is only one reason, albeit an important one, for doing this. The tools and techniques adequate to handle portability problems are applicable in a broader context, i.e. whenever there is a family of related programs which must be developed and evolved together.

To elaborate these observations, let us begin by looking at the problem of portability of mathematical software. There are several questions we must try to answer.

1. What is mathematical software?

Frequently, in discussions about mathematical software, an item of mathematical software is thought of as the implementation of a single algorithm. The only larger entities considered are libraries, i.e. collections of implementations of often quite unrelated algorithms, presented in a unified and consistent manner.
For our purposes here, we will broaden the definition to include also scientific and engineering codes. These codes are large programs that provide the mechanism to solve any computational problem in some application area. The mathematical theory on which such a code is based is typically quite specific to the application area. Examples of such codes are NASTRAN, a code for solving structural engineering problems; XTAL, a code for crystallographic problems; or GENSTAT, a code for performing statistical analyses.

Including these codes in our definition of mathematical software accentuates the difficulties associated with families of programs because of the sheer complexity of these codes. They often are 20,000 or more lines of source text in a high-level language. They often are structured as 250 or more subroutines. They typically have elaborate languages for computational control and data input, combined with flexible options for output and data display - and together these "nonmathematical" parts can make up as much as 80% of the source text for the program. These codes usually interact with stored data structures in the file system of the host environment.

Codes of this kind represent a considerable investment. Often they are built over periods of 10 years or more, so they outlast generations of computers. They contain considerable subject area knowledge beyond the mathematical algorithms. The mathematical algorithms they do contain are often specifically developed for use in these codes. The arguments for the importance of tools and techniques for managing and evolving families of programs are particularly strong with respect to these codes.

2. Whose needs must the tools and techniques address?

In designing tools and techniques for working with families of programs, it is essential to understand how and by whom they will be used, and what aspects will be important. Four situations are immediately apparent.

The first situation is where there are researchers working in the area covered by an item of mathematical software, developing new algorithms. In the initial development stages, they need to perform experiments, modifying the given item of mathematical software to utilize their new methods, or to support detailed instrumentation. The customized versions will probably never be used by anyone except the researcher himself, and typically have a short lifetime. The primary need to be met is that such experimental customization is not just feasible but cheap.