Users' Guide to NetSolve

version 1.2.beta

`http://www.cs.utk.edu/netsolve`

Henri Casanova[1] Jack Dongarra[1 2] Alexander Karainov [3] Jerzy Wasniewski[3]

October 27, 1998

[1]Department of Computer Science, University of Tennessee, TN 37996
[2]Mathematical Science Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831
[3]Danish Computing Centre for Research and Education (UNI•C), DTU, Bldg. 304, DK-2800 Lyngby, Denmark

Abstract

The NetSolve system, developed at the University of Tennessee, is a client-server application designed to solve computational science problems over a network. This document is organized in eight chapters. The first chapter presents the basic concepts of NetSolve and gives references to several publications concerning the project. The second chapter describes the specification of NetSolve problems. The third chapter describes how to download, install and test the NetSolve software. The fourth chapter describes the NetSolve client interfaces that are provided with the package (Matlab, Mathematica, Java, C and Fortran APIs and the Java GUI). The fifth chapter describes a new feature of NetSolve: request farming. The sixth chapters gives details about the agent/server side of NetSolve, especially how to customize a NetSolve server. The seventh and last chapter describes how NetSolve handles *user-provided functions*. Finally, complete reference manuals as well as complete API examples are found in appendices.

Contents

Chapter 1

The NetSolve System

1.1 Introduction

The efficient solution of large problems is an ongoing thread of research in scientific computing. Various mechanisms have been developed to perform computations across diverse platforms. The most common mechanism involves software libraries. Unfortunately, the use of such libraries presents several difficulties. Some software libraries are highly optimized for only certain platforms and do not provide a convenient interface to other computer systems. Other libraries demand considerable programming effort from the user. While several tools have been developed to alleviate these difficulties, such tools themselves are usually available on only a limited number of computer systems and are rarely freely distributed. Matlab [1] or Mathematica [2] are examples of such a tools.
These considerations motivated the establishment of the NetSolve project. The basic philosophy behind NetSolve is to provide a uniform, portable and efficient way to access computational resources over a network. NetSolve is a client-server application, and a number of different client interfaces have been developed to the NetSolve software. Users of C, Fortran, Matlab, Mathematica, Java or the World Wide Web can easily use NetSolve.

1.2 Overview of the NetSolve System Architecture

The NetSolve system is a set of loosely connected machines. By *loosely* connected, we mean that these machines can be on the same local network or on an international network, and administrated by different institutions and organizations. Moreover, the NetSolve system can be running in a *heterogeneous* environment, which means that machines with different internal data representations can be in the system at the same time.
Figure 1.1 shows the global conceptual picture of the NetSolve system. In this figure, we can see the three major components of the system:

- The NetSolve client,
- The NetSolve agent,
- The NetSolve computational resources (or servers).

Conceptually, solving a problem with NetSolve is done in four steps: (i) the client sends a request to the agent; (ii) the agent chooses the "best" NetSolve resource according to the size and nature of the problem to be solved; (iii) the problem is solved on the chosen server; (iv) and the result is sent back to the client.

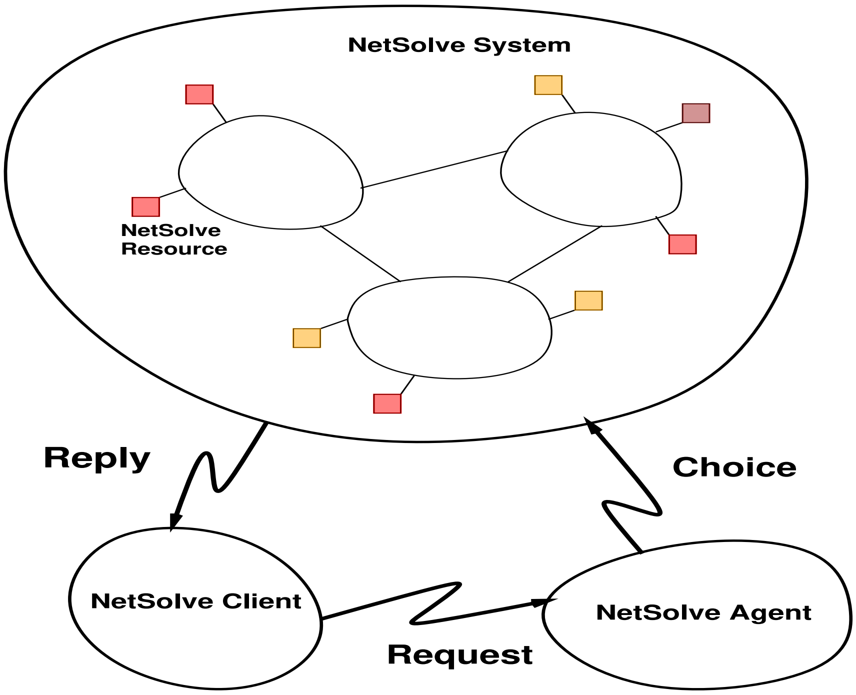

Figure 1.1: The NetSolve System

The system is fault tolerant, meaning that fault-recovery is transparent to the end user and that an answer to a request is returned unless every resource in the system has failed or is unavailable. The NetSolve agent is the key to the load-balancing strategy, and details about its design can be found in [3]. Recent developments and applications of NetSolve are described in [4]. Lastly, technical details about NetSolve's implementation are to be found in [5].
Figure 1.2 shows the new software hierarchy of NetSolve version 1.2.

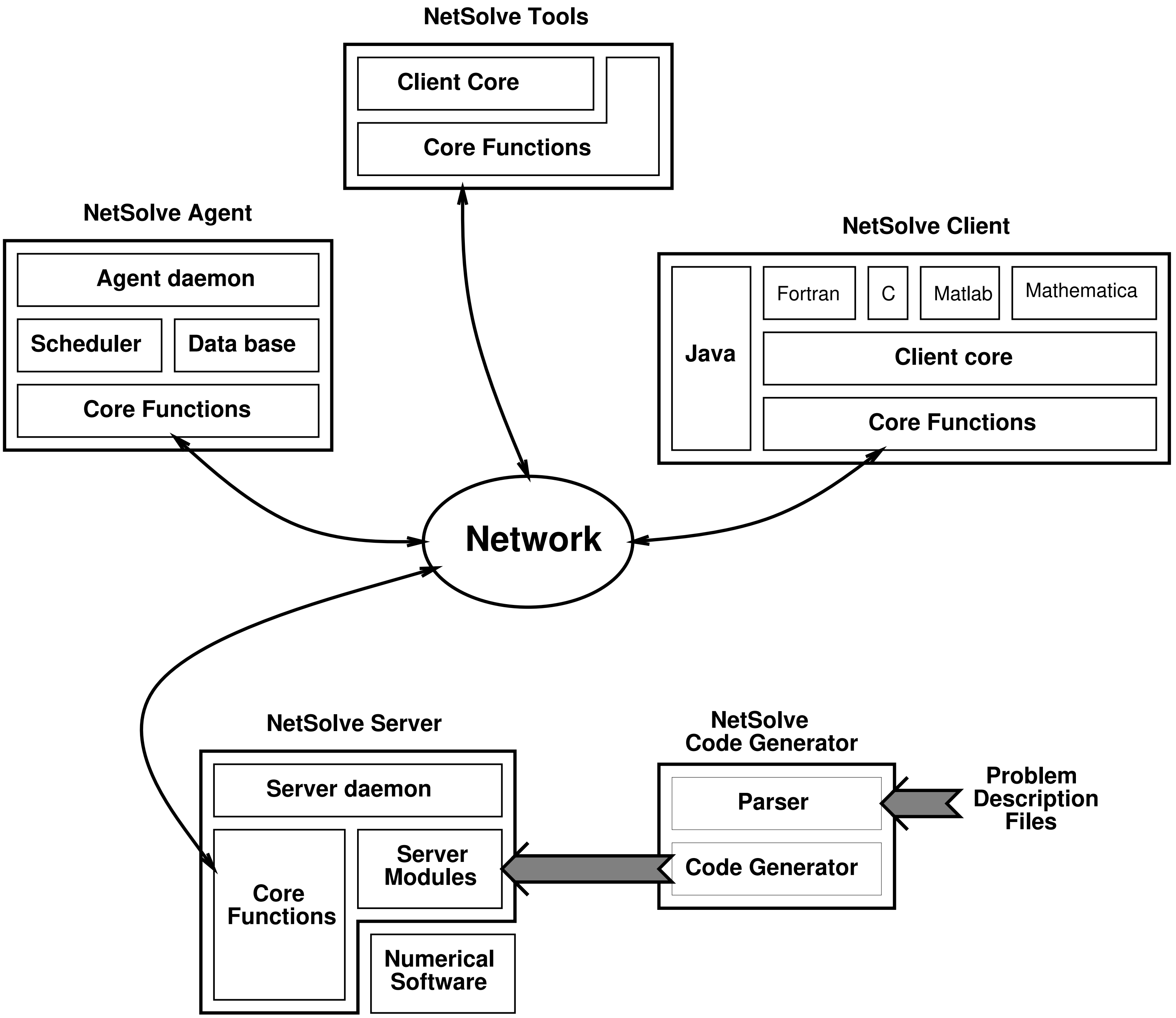

Figure 1.2: NetSolve's internal software hierarchy

Chapter 2

NetSolve Problem Specification

2.1 Object and Data type

Solving a computational problem with NetSolve is really a function evaluation:

$$< output >=< name > (< input >)$$

where

- $< name >$ is a character string containing the name of the problem,
- $< inputs >$ is a list of input objects,
- $< outputs >$ is a list of output objects.

An object is itself described by an *object type* and a *data type*. The types available in the current version of NetSolve are shown in table 2.1. Rather than giving here examples for each object type, we refer the reader to the programs in: `NetSolve-1.2.x/src/Example` and `NetSolve-1.2.x/src/Testing`.

2.2 Relevance to the different user Interfaces

In the previous section, we described the input and output of a NetSolve problem as lists of *objects*. The Matlab, Mathematica, and Java interfaces to NetSolve can manipulate objects directly and it is therefore very easy to call NetSolve from there interfaces once problem descriptions are known. From interfaces that are not object-oriented (C and Fortran), it is necessary to use a *calling sequence* that describes the objects' features individually. Let us illustrate this with an example. Let us suppose that there is a NetSolve problem named *foo* that takes an integer vector in input and return the sum of its components. A call to that problem from the Matlab interface would look like:

```
x = rand(10,1)    // creates a random vector of size 10
sum = netsolve('foo',x)
```

where `x` is the input vector. A call from the C interface on the other hand is more complex:

```
netsolve('foo()',x,10,&sum);
```

Indeed, the call to the C interface has to pass the size of the vector (10 is the example), whereas that size is part of the Matlab vector object. It is also clear that there can be several calling sequences from the C (or Fortran) interface:

Data Type	Description	Note
NETSOLVE_I	Integer	
NETSOLVE_CHAR	Character	
NETSOLVE_BYTE	Byte	never XDR encoded
NETSOLVE_FLOAT	Single precision real	
NETSOLVE_DOUBLE	Double precision real	
NETSOLVE_SCOMPLEX	Single precision complex	
NETSOLVE_DCOMPLEX	Double precision complex	
Object Type	**Description**	**Note**
NETSOLVE_SCALAR	scalar	
NETSOLVE_VECTOR	vector	
NETSOLVE_MATRIX	matrix	
NETSOLVE_FILE	file	only of data type NETSOLVE_CHAR
NETSOLVE_PACKEDFILES	packed files	only of data type NETSOLVE_CHAR
NETSOLVE_UPF	User Provided Function	only of data type NETSOLVE_CHAR
NETSOLVE_STRING	Character string	only of data type NETSOLVE_CHAR
NETSOLVE_STRINGLIST	Character string list	only of data type NETSOLVE_CHAR

Table 2.1: Available data and object types

```
netsolve('foo()',&sum,10,x);
netsolve('foo()',&sum,x,10);
```

could be valid calls. In fact, NetSolve servers impose (see Section 6.3.3) a given calling sequence to users of the non-object-oriented interfaces. There are several other differences between the Matlab and C calls and the specifics are given in Section 4.2 and 4.3. Section 4.3 also contains an entire example, including the discovery of a problem by the user and more importantly of its calling sequence from C or Fortran.

2.3 Problems that can be solved with NetSolve

Before actually using NetSolve with any interface, the user needs to know what problems are solvable. It is possible to query NetSolve agents in order to obtain a list and descriptions of the problems that can be solved by the servers known to that agent. There are several ways of sending such queries.

First, from the NetSolve homepage, it is possible to specify an agent name and run CGI scripts to obtains detailed informations about NetSolve problems, including C and Fortran calling sequences specifications. Problem lists and descriptions are also directly available from the Matlab interface, the Mathematica interface, and the Java GUI. Lastly, the tools described in Section 6.4 also give access to that information from the UNIX prompt.

The full name of a problem has two parts: (i) the *path*, and (ii) the *nickname*. Let us demonstrate this with an example. The problem nicknamed `ddot`, which computes the inner product of two double-precision vectors, can have full name `/BLAS/Level1/ddot`. This full name has two purposes. First, when we display a list of problems, they are sorted alphabetically on their full name, and the problems are grouped by "directory". Second, by convention, the first element of the full name (e.g., `BLAS`) is the name of the numerical library the problem uses. This convention has proven to be useful, as seen in Section 4.3.2.

Chapter 3

Downloading, Installing, and Testing the Software

3.1 Downloading

The software can be downloaded from the NetSolve homepage and the following UNIX commands will create the `Netsolve-1.2.x` directory, where `x` is the latex sub-version number:

```
% gunzip NetSolve-1.2.x.tar.gz
% tar -xf NetSolve-1.2.x.tar
```

3.2 Setting the NetSolve Root

It is necessary to set an environment variable that contains the path to the `Netsolve-1.2.x` directory. It should be defined in the `.cshrc` as:

```
setenv NETSOLVE_ROOT /home/me/Netsolve-1.2.x  (for csh)
```

or,

```
export NETSOLVE_ROOT=/home/me/Netsolve-1.2.x  (for ksh)
```

for instance. From now on, we assume that the Shell is from the `csh` family.

3.3 Setting the Architecture

In addition to the `NETSOLVE_ROOT` variable, one must also set a variable that contains the architecture of the platform for which the software is going to be compiled. The `Netsolve-1.2.x` directory contains a Shell script called `netsolvegetarch` that can be used to return a character string describing the architecture of the machine of the user. Suppose, for instance, that one wishes to run the script on an IBM RS/6000:

```
% netsolvegetarch
  RS6K
```

In that case, the `NETSOLVE_ARCH` environment variable should be defined in the `.cshrc` file as:

```
setenv NETSOLVE_ARCH RS6K
```

Preferably, one can set the variable in a platform-independent fashion:

```
setenv NETSOLVE_ARCH `$NETSOLVE_ROOT/netsolvegetarch`
```

To date, NetSolve has been fully ported to the architectures in table 3.1. In addition, the NetSolve distribution also contains Matlab and C APIs for Win32.

`$NETSOLVE_ARCH`	Platform
`SUN4`	Sun 4, 4c, SPARC, etc.
`SUN4SOL2`	Sun 4 running Solaris 2.x
`ALPHA`	DEC Alpha/OSF-1
`PMAX`	DEC Pmax running NetBSd
`NEXT`	NeXT
`SGI5`	Silicon Graphics IRIX running OS $\geq$ 5.0
`SGI64`	Silicon Graphics IRIX running OS $\geq$ 6.0
`HPPA`	HP 9000 PA-Risc
`RS6K`	IBM RS/6000
`LINUX`	Linux
`SCO`	FreeBS
`T3E`	Cray T3E

Table 3.1: UNIX platforms supported by NetSolve

3.4 Compiling

Now that the `NETSOLVE_ARCH` and `NETSOLVE_ROOT` environment variables have been set as described in Section 3.3, the software can be compiled.

3.4.1 Customizing the Compilation

Even though suitable default options are provided for the compilation of the software, one may look in the `Netsolve-1.2.x/conf` directory to edit the `conf.def` file. This file contains custom parameters about the NetSolve installation, for example:

```
#
# directories
#
OBJDIR          =  $(NETSOLVE_ROOT)/obj/$(NETSOLVE_ARCH)
XDIR            =  $(NETSOLVE_ROOT)/bin/$(NETSOLVE_ARCH)
LIBDIR          =  $(NETSOLVE_ROOT)/lib/$(NETSOLVE_ARCH)
MATLABOBJDIR    =  $(OBJDIR)/Matlab

# verbosity
#
#       DEBUG : For total debugging information
#       VIEW  : For smooth information during the execution
#       NONE  : No visual output
DEBUGING = VIEW
```

```
# Globus Heart Beat Monitor
#
#HBMLM     =  <path to the hbmlm>
#HBMDC     =  <path to the hbmdc>
#HBMDCPORT =  <port of the hbmdc>
#HBMLIB    =  <hbm library>
#HBMINCDIR =  <hbm includes directories>
#HBMFLAG   = -I$(HBMINCDIR) -DHBMLM=$(HBMLM) -DHBMDC=$(HBMDC) -DHBMDCPORT=$(HBMDCPORT)

# MPI
#
MPI_INCDIR = /src/icl/MPI/mpich1.1/include
```

Let us review the four elements of the custom installation in this version of NetSolve:

Directories : The first section of the `conf.def` file contains the name of the directories where the object files, executables, libraries, and special Matlab object files should be generated.

Verbosity : This section specifies how much output should be generated on the screen when running agents, servers, or clients.

Globus Heart Beat Monitor : If the 6 variables in the third section are uncommented and assigned the appropriate values, then NetSolve uses the Globus HeartBeat Monitor for failure detection. The Globus project homepage is located at:

`http://www.globus.org`,

where the software and documentation are available.

MPI : In the case of a parallel server (see Section 6), it is necessary to set the `MPI_INCDIR` variable to the path to the `"mpi.h"` file.

The files `$NETSOLVE_ROOT/conf/$NETSOLVE_ARCH.def` contain settings for compilers/linkers/system libraries for the different platform and should be modified only in the case of compilation problems.

3.4.2 Typical Compiling

Typing `make` in the `Netsolve-1.2.x` directory will give instructions to complete the compilation. Typical compilations include:

```
make standard
```

builds the agent, server, the NetSolve test suite (see Section 3.5) and examples,

```
make C_client Fortran_client Matlab mathematica java
```

builds all the client interfaces provided in the NetSolve distribution, and

```
make tool
```

builds the NetSolve management tools (see Section 6.4).

3.5 Testing the Software

Testing the software consists in starting an agent and a server and running a client test (Section 3.5.1). Alternatively, the default agent and servers running at the University of Tennessee can be used to test the client only (see Section 3.5.2). We describe here the step-by-step procedure that involves manipulations that will be detailed and explained in the following chapters. We assume that the Shell is `csh`.

3.5.1 Agent-Server-Client Test

1. Choose a machine to run the agent, server and client (say `netsolve.net.world`),
2. On that machine, set the two environment variables of Sections 3.2 and 3.3,
3. `cd $NETSOLVE_ROOT`,
4. edit the file `./server_config`,
5. replace `bluegrass.cs.utk.edu` by `netsolve.net.world`,
6. save the file,
7. `make agent server test`
8. start three `xterm` sessions.
9. In the first `xterm`:
   (a) `cd $NETSOLVE_ROOT/bin/$NETSOLVE_ARCH`
   (b) `agent`
10. In the second `xterm`:
    (a) `cd $NETSOLVE_ROOT/bin/$NETSOLVE_ARCH`
    (b) `server`
11. In the third `xterm`, once the server is running:
    (a) `setenv NETSOLVE_AGENT netsolve.world.net`
    (b) `cd $NETSOLVE_ROOT`
    (c) `Test`

While the test suite is running, it prints out messages about its execution. This test tests only the C and Fortran clients. See Chapters 4.2, 4.4, and 4.6 for details on how to test the other interfaces.

3.5.2 Client Test

1. `cd $NETSOLVE_ROOT`
2. `make test`
3. `setenv NETSOLVE_AGENT bluegrass.cs.utk.edu`
4. `Test`

While the tester is running, it prints out messages about its execution. This test tests only the C and Fortran clients. See Sections 4.2, 4.4, and 4.6 for details on how to test the other interfaces.

Chapter 4

Provided NetSolve Client Interfaces

4.1 Getting Started

4.1.1 Setting an Agent Name

As described in Section 1.2, to solve a problem a client must contact an agent. The C, Fortran, Matlab, and Mathematica interfaces require the environment variable `NETSOLVE_AGENT` to be set to contain the name of a host running a NetSolve agent. If the user knows of a NetSolve system installed somewhere, he will have to ask the NetSolve administrator for the name of such a host. The NetSolve homepage includes a list of registered agents on the Internet. The constantly running agent at the University of Tennessee is `bluegrass.cs.utk.edu`. If the user wishes to set his agent to be this one, he will have to modify his `.cshrc` as follows:

```
setenv NETSOLVE_AGENT bluegrass.cs.utk.edu
```

Note that the Java GUI and API do not require the `NETSOLVE_AGENT` environment variable to be set. Instead of specifying a hostname, one can also specify an IP address.

4.2 Matlab Interface

4.2.1 Introduction

Building the Matlab interface by typing

```
make matlab
```

in the directory `NetSolve-1.2.x` produces the four following *mex-files* :

- `Netsolve_client/bin/$NETSOLVE_ARCH/netsolve.mex###`
- `Netsolve_client/bin/$NETSOLVE_ARCH/netsolve_nb.mex###`
- `Netsolve_client/bin/$NETSOLVE_ARCH/netsolve_err.mex###`
- `Netsolve_client/bin/$NETSOLVE_ARCH/netsolve_errmsg.mex###`

The `###` part of the extension depends on the architecture (for instance, the extension is `.mexsol` for the Solaris Operating System). These four files alone are the Matlab interface to NetSolve. Modifying the `MATLABPATH` environment variable as

```
setenv MATLABPATH $NETSOLVE_ROOT/bin/$NETSOLVE_ARCH
```

will make these four files accessible to Matlab. It is also possible to use the Matlab command `addpath`. For more information about mex-files, the user can refer to [1]. Basically, the user will now be able to call four new functions from Matlab: `netsolve()`, `netsolve_nb()`, `netsolve_err()`, and `netsolve_errmsg()`. The following sections explain how to use those functions.

4.2.2 What to Do First

Let us now assume that the user has compiled the Matlab interface, set an agent name, started a Matlab session and is now ready to try NetSolve. In this section we describe those features of the interface that allow the user to get information about the currently available NetSolve system.
As stated briefly in Section 2.3, it is possible to obtain the list of solvable problems from Matlab. Let us try that first:

```
>> netsolve
NetSolve - List of problems available -
/BLAS/Matrices/matmul
/ItPack/jsi
/LaPack/Matrices/EigenValues/eig
/LaPack/Matrices/SingularValues/svd
>>
```

Every line contains a full problem name. Let us now assume that the user is wondering about what kind of problem `eig` is. He can type

```
>> netsolve('eig')
```

This command will provide detailed information about this particular problem. Let us split the output produced by this command into different pieces:

```
-- eig --  From LAPACK -
Simplified version
Computes the eigenvalues of a double-precision real
square matrix A. Returns two double-precision real
vectors containing respectively the real parts and
the imaginary parts of the eigenvalues.

Matlab Example : [r i] = netsolve('eig',a)
```

The same kind of information is also available from the homepage CGI scripts or the management tools. It gives a short description of what the problem is. In this case, it also includes an example for Matlab, using `netsolve()`.

```
* 1 objects in INPUT
 - input 0: Matrix Double Precision Real.
 Matrix A
```

This is the description of the input the user needs to give NetSolve. This particular problem requires only one double-precision matrix in input. Notice that this matrix has to be square (as stated in the description of the problem). If the user tries to call NetSolve for this problem with a rectangular matrix, he will receive an error message stating that the dimensions of the input are invalid.

```
* 2 objects in OUTPUT
```

```
 - output 0: Vector Double Precision Real.
 Real parts of the eigen values
 - output 1: Vector Double Precision Real.
 Imaginary parts of the eigen values
```

The output of the problem is described here. The problem `eig` will return two vectors, the real and imaginary parts of the eigenvalues of the input matrix, respectively.

```
-----------------------------------------------
Output Objects 0 and 1 can be merged.
```

This last part does not appear for every problem and is relevant only for the Matlab interface. Since Matlab provides a mechanism to manipulate complex objects, it is probable that the user would like to have `eig` return one single complex vector instead of two separate real vectors. This point is further developed in the next section.

The Matlab interface has another feature that is concerned not with the actual problem solving but with providing information about the NetSolve configuration itself. We have just seen how to get information about the problems handled by the NetSolve servers; it is also possible to obtain the physical locations of these servers. Let us assume that our `NETSOLVE_AGENT` environment variable is set to `bluegrass.cs.utk.edu` (see Section 4.1.1). Let us try the following command:

```
>> netsolve('?')
```

this command produces the following output :

```
NetSolve - List of available agents -
bluegrass.cs.utk.edu(128.169.92.78)
NetSolve - List of available servers -
maruti.cs.berkeley.edu(128.32.36.83)
cupid.cs.utk.edu(128.169.94.221)
```

We can see that there are three servers in the NetSolve system containing the machine `bluegrass` at the University of Tennessee:

1. `bluegrass` itself, which runs an agent,
2. `cupid` at the same location, which runs a server,
3. `maruti` at U.C. Berkeley, which is also a computational server.

The same information can be obtained from the homepage CGI scripts or the management tools.

4.2.3 Calling `netsolve()` to perform computation

The simplest way to perform an actual numerical computation is to call the function `netsolve()`. With this function, the user can send a blocking request to NetSolve. By *blocking* we mean that after typing the command in the Matlab session, the user gets back control only when the computation has been successfully completed on a server. The other way to perform computation is to send a nonblocking request as described in Section 4.2.4.

Let us go on with the `eig` example we started to develop in the preceding section. The user now knows that he has to provide a double-precision square matrix to NetSolve, and he knows that he is going to get two real vectors back (or one single complex vector). He first creates a 300×300 matrix, for instance,

```
>> a = rand(300);
```

The call to NetSolve is now

```
>> [x y] = netsolve('eig',a)
```

All the calls to `netsolve()` will look the same. The left-hand side must contain the output arguments, in the same order as listed in the *output description* (see Section 4.2.2). The first argument to `netsolve()` is always the name of the problem. After this first argument the input arguments are listed, in the same order as they are listed in the *input description* (see Section 4.2.2). This function does not have a fixed calling sequence, since the number of inputs and outputs depends on the problem the user wishes to solve.
Let us see what happens when we type:

```
>> [x y] = netsolve('eig',a)
Contacting server on cupid.cs.utk.edu

x =          y =
    10.1204             0
    -0.9801        0.8991
    -0.9801       -0.8991
    -1.0195             0
    -0.6416        0.6511
      ...           ...
      ...           ...
```

As mentioned earlier, the user can decide to regroup x and y into one single complex vector. Let us make it clear again that this possibility is a specificity of `eig` and is not available in general for all problems. To merge x and y, the user just has to type:

```
>> [x] = netsolve('eig',a)
Contacting server on cupid.cs.utk.edu

x =
    10.1204
    -0.9801 + 0.8991i
    -0.9801 - 0.8991i
    -1.0195
    -0.6416 + 0.6511i
        .........
        .........
```

4.2.4 Calling `netsolve_nb()`

The obvious drawback of the function `netsolve()` is that while the computation is being performed remotely, the user must simply wait to get back the prompt. To address this drawback, we provide `netsolve_nb()`. This second function allows the user to send nonblocking requests to NetSolve. Once the user has called `netsolve_nb()`, he gets back the control. He can then do some work in *parallel* and check for the completion of the request later. He can even send multiple requests to NetSolve. Thanks to the load-balancing strategy implemented in the NetSolve agent, all these requests will be solved on different machines if possible, achieving some *NetSolve-parallelism*. Let us now describe this function on the `eig` example.
As in Section 4.2.3, the user creates a 300×300 matrix and calls NetSolve:

```
>> a = rand(300);
>> [r] = netsolve_nb('send','eig',a)
```

Obviously, the calling sequence to `netsolve_nb()` is a little different from the one to `netsolve()`. The left-hand side always contains one single argument. Upon completion of this call, it will contain a *NetSolve request handler*. The right-hand side is composed of two parts: the *action* to perform and the arguments that would be passed to `netsolve()`. In this example, the action to perform is `'send'`, which means that we send a request to NetSolve. Throughout this section, we will encounter all the possible actions, and they will be summarized in Appendix A.

Let us resume our example and see what NetSolve answers to the first call to `netsolve_nb()` :

```
>> [r] = netsolve_nb('send','eig',a)
Contacting server cupid.cs.utk.edu
r = 0
```

`netsolve_nb()` returns a request handler: `0`. This request handler will be used in the subsequent calls to the function. The request is being processed on `cupid`, and the result will eventually come back. The user can obtain this result in one of two ways. The first one is to call `netsolve_nb()` with the `'probe'` action :

```
>> [status] = netsolve_nb('probe',r)
```

`netsolve_nb()` returns the *status* of a pending request. The right-hand side contains the action, as is required for `netsolve_nb()`, and the request handler. This call returns immediately, and prints out a message. Here are the two possible scenarios:

```
>> [status] = netsolve_nb('probe',r)
Not ready yet
status = -1
...
>> [status] = netsolve_nb('probe',r)
Result available
status = 0
```

To obtain the result of the computation one must call `netsolve_nb()` with the `'wait'` action:

```
>> [x y] = netsolve_nb('wait',r)
x =          y =
    10.1204             0
    -0.9801        0.8991
    -0.9801       -0.8991
    -1.0195             0
    -0.6416        0.6511
      ...            ...
      ...            ...
```

As for `netsolve()`, one can merge the real part and the imaginary part into a single complex vector. The typical scenario is to call `netsolve_nb()` with the action `'send'`, then make repeated calls with the action `'probe'` until there is nothing more to do than wait for the result. The user then calls `netsolve_nb()` with the action `'wait'`. It is of course possible to call `netsolve_nb()` with the action `'wait'` before making any call with the action `'probe'`. One last action can be passed to `netsolve_nb()`, as shown here:

```
>> netsolve_nb('status')
```

This command will return a description of all the pending requests. Let us see how it works on this last complete example:

```
>> a = rand(800); b = rand(800);
```

```
>> [r1] = netsolve_nb('send','eig',a)
Contacting server on cupid.cs.utk.edu
r1 = 0
>> [r2] = netsolve_nb('send','eig',b)
Contacting server on vw.cs.Berkeley.edu
r2 = 1
```

Now let us see what `status` does:

```
>> netsolve_nb('status')
Pending NetSolve requests :
Request #0: 'eig', submitted to cupid.cs.utk.edu(128.169.94.221)
        Result available
Request #1 - eig
        Assigned to vw.cs.Berkeley.edu 3 seconds ago
        Should complete in 202 seconds
```

The user can check what requests he has sent so far and obtain an estimation about the completion times. By using the `status` action, the user can also find out whether a request is still running or has been completed. By sending multiple non blocking requests to NetSolve and relying on the agent for load balancing, the user can achieve parallelism.

4.2.5 What Can Go Wrong?

During a computation, two classes of error can occur: NetSolve failures and user mistakes. Let us show here a few examples:

```
>> netsolve
No agent running on demidoff.cs.utk.edu
...
>>[x] = netsolve('foo',a)
Unknown problem 'foo'
...
>> [a b] = netsolve_nb('send','eig',a)
Matrix 'a' should be square
```

In case of error, the different NetSolve functions print out appropiate error messages. However, when the user writes Matlab scripts that call NetSolve, he/she needs Ways to catch the errors while the script is running. Hence the functions described in the next section.

4.2.6 Catching NetSolve errors

There are two NetSolve functions that can be called from Matlab to catch errors. The first function, `netsolve_err()` takes no arguments and returns an integer that is the NetSolve error code returns by the last call to a NetSolve function (see Apppendix D for a list of the possible error codes). Here is a call:

```
>> e = netsolve_err
e = -4
```

The other function, `netsolve_errmsg()` takes an error code as argument and returns a string that contains the corresponding error message. It is usual to call `netsolve_errmsg()` as follows:

```
>> [msg] = netsolve_errmsg(netsolve_err)
msg = Impossible to contact Agent
```

With these two functions, it is possible to write Matlab scripts that call NetSolve and handle all the NetSolve errors at runtime.

4.2.7 Demo

A NetSolve-Matlab demo is available with the NetSolve distribution. It consists of a set of Matlab scripts that call NetSolve to compute parts of the Mandelbrot set. The main script is called `mandel.m` and is located in `$NETSOLVE_ROOT/src/Demo/mandelbrot/`. To run the demo, just type `mandel` at the Matlab prompt.

4.3 C and Fortran Interfaces

4.3.1 Introduction

The Fortran interface is built on top of the C interface, since all the networking underneath NetSolve has to be done in C. However, we chose to write the Fortran interface with subroutines instead of functions (for reasons of compiler compatibilities). The C functions all return an NetSolve *error code* equal to `0` in case of success or to a negative value in case of error. Appendix D contains the list of all possible error codes. The Fortran subroutines just take an extra integer argument (passed by reference) that contains the error code after completion of the call. The reference manuals for C and Fortran are in Appendices B and C.
The basic concepts here are the same as the ones we have introduced in Section 4.2 for the Matlab interface, especially the ability to call NetSolve in a blocking or nonblocking fashion.
Compiling the C/Fortran interface as explained in Section 3.4 produces the two following archive files:

- `Netsolve-1.2.x/lib/$NETSOLVE_ARCH/libnetsolve.a` : the C library
- `Netsolve-1.2.x/lib/$NETSOLVE_ARCH/libfnetsolve.a` : the Fortran library

Before linking to one of those libraries, the user must include the appropriate header file in his program:

- `Netsolve-1.2.x/include/netsolve.h` in C,
- `Netsolve-1.2.x/include/fnetsolve.h` in Fortran.

The Fortran file inclusion is not mandatory, but simply allows calling subroutines to manipulates the NetSolve error codes by variable name rather than by variable value to increase source readability.
We describe the C and Fortran interfaces by the mean of an examples. In the following section we start developing the example by demonstrating how a user can et information about the calling sequence to a given problem.

4.3.2 What Calling Sequence ?

In Section 2.2 we already described the reasons why the C and Fortran interfaces, as they are not object-capable, need to use specific calling sequences that are more involved than the ones used from Matlab or Mathematica .
Let us take a very simple example: the user wants to perform a dense linear system solve. The first thing to know, as stated in earlier chapters, is the name or IP address of a host running a NetSolve agent. The default NetSolve agent running at the University of Tennessee is aware of many servers that can perform the computation. In fact, a dense linear system solve is provided with the NetSolve distribution as a default numerical software for the server. The user has now two possible courses of action to find out about the problem (see Section 2.3). Let us assume that the user chooses to use the UNIX command line management tools (see Section 6.4 for a complete description of these tools). The alternative would be to use the CGI scripts on the NetSolve homepage. Here is a typical session:

```
% NS_problems bluegrass.cs.utk.edu
/ImageProcessing/Filters/blur
/LinearAlgebra/LAPACK/LinearSystems/dgesv
/LinearAlgebra/LAPACK/LinearSystems/linsol
% NS_probdesc bluegrass.cs.utk.edu dgesv
-- dgesv -- From LAPACK -
Compute the solution to a real system of linear equations
  A * X = b
where A is an N-by-B matrix and X and B are N-by-NRHS matrices.
Matlab Example : [x y z info ] = netsolve('dgesv',a,b)
http://www.netlib.org/lapack/index.html
* 2 objects in INPUT
 - input 0: Matrix Double Precision Real.
 Matrix A
 - input 1: Matrix Double Precision Real.
 Right hand side
* 4 objects in OUTPUT
 - output 0: Matrix Double Precision Real.
 LU factors ( A = P*L*U)
 - output 1: Vector Integer.
 Vector of pivots (defines the P matrix)
 - output 2: Matrix Double Precision Real.
 Solution
 - output 3: Scalar Integer.
 INFO
  0  successful
  <0 error on calling ?
  >0 QR algorithm failed
...[to be continued]...
```

We will see in Chapter 6.3 how the servers specify the calling sequence to a given problem. It is usual for servers to enforce the same calling sequence as the original numerical software and to give a problem the name of the original library function. In the example, `dgesv()` is the name of a LAPACK subroutine and the user can therefore expect the calling sequence for the problem `dgesv` to match the one of the subroutine. One can see in the problem list returned by `NS_problems` a problem called `linsol`. In this very example, `linsol` is a simplified version of `dgesv` and has a simplified calling sequence chosen by whoever started the first server that provides access to that problem. Since `linsol` is not the name of a LAPACK subroutine, that calling sequence can be arbitrary.

Now, two situations are possible. First, the user already knows the numerical software (here LAPACK) and may even have code already written in terms of this software. Then, *switching* to NetSolve is immediate. The second possibility is that the user does not know the software. Then he needs to pay attention to the end of the output given by `NS_probdesc`:

```
* Calling sequence from C or Fortran
8 arguments
 - Argument #0:
   - number of rows of input object #0 (A)
   - number of columbs of input object #0 (A)
   - number of rows of input object #1 (RHS)
 - Argument #1:
   - number of columbs of input object #1 (RHS)
 - Argument #2:
```

```
   - pointer to input object #0 (A)
   - pointer to output object #0 (LU)
   - pointer to output object #0 (LU)
 - Argument #3:
   - leading dimension of input object #0 (A)
 - Argument #4:
   - pointer to output object #1 (PIVOT)
 - Argument #5:
   - pointer to input object #1 (RHS)
   - pointer to output object #1 (PIVOT)
   - pointer to output object #2 (SOLUTION)
 - Argument #6:
   - leading dimension of input object #1 (RHS)
 - Argument #7:
   - pointer to output object #3 (INFO)
```

This output can appear rather cryptic at first. Let us work through it step by step. First, the number of arguments in the calling sequence is 8. This means that the call from C will look like:

```
status = netsl('dgesv()',X0,X1,X2,X3,X4,X5,X6,X7);
```

Now, each argument is described in the output of `NS_probdesc` and this description can be translated into meaningful variable names in the user source code. For instance, `X2` should be a pointer to the matrix of the linear system, and `X3` should be an integer that is the leading dimension of the matrix. We can now move on to the descriptions of the different ways of calling NetSolve from C or Fortran.

4.3.3 Blocking Call

The blocking call to NetSolve from C or Fortran is the simplest. Specifically, one calls a single function, `netsl()`. This function returns an error code. It takes as arguments the name of a problem and the list of input data. These inputs are listed according to the calling sequence discussed in Section 4.3.2. The C prototype of the function is

```
int netsl(char *problem_name, ... < argument list > ...)
```

and the Fortran prototype is

```
SUBROUTINE FNETSL(PROBLEM_NAME, INFO, ... < argument list > ...)
```

where `PROBLEM_NAME` is a string and `INFO` is the status code returned by NetSolve.
Let us resume our example of the call to `dgesv`. In Fortran, the direct call to LAPACK looks like

```
      call DGESV(N,1,A,MAX,IPIV,B,MAX,IERR)
```

The equivalent blocking call to NetSolve is

```
      call FNETSL('DGESV()',INFO,
     $          N,1,A,MAX,IPIV,B,MAX,IERR)
```

The call in C is

```
info = netsl('dgesv()',n,1,a,max,ipiv,b,max,&info)
```

Notice that the name of the problem is *case insensitive* and that it is postfixed by an opening and a closing parenthesis. The parentheses are used by NetSolve to handle Fortran/C interoperability on certain platforms.

In Fortran, every identifier represents a pointer, but in C we actually had the choice to use pointers or not. We chose to use integer (`int`) for the sizes of the matrices/vectors, but pointers for everything else.
From the user's point of view, the call to NetSolve is exactly equivalent to a call to LAPACK. One detail, however, needs to be mentioned. Most numerical software is written in Fortran and requires users to provide workspace arrays as well as data, since there is no possibility for dynamic memory allocation. Because we preserved the exact calling sequence of the numerical softwares, we require the user to pass those arrays. But, since the computation is performed remotely, workspace on the client side is meaningless. It will, in fact, be dynamically created on the server side. Therefore, when the numerical software would require workspace, the NetSolve user may provide an empty workspace! This is signaled in the output of `NS_probdesc` by an argument description such as:

```
- Argument #6:
  - ignored
```

4.3.4 Nonblocking Call

We developed this nonblocking call for the same reason we developed one for Matlab (see Section 4.2.4): to allow the user to have some *NetSolve-parallelism.* The nonblocking version of `netsl()` is `netslnb()`. The user calls it exacly as he would call `netsl()`. If the call to `netslnb()` is successful, it returns a request handler in the form of an (positive) integer. If it is not successul, it returns an error code. Continuing with our example:

```
      call FNETSLNB('DGESV()',REQUEST,
                N,1,A,MAX,IPIV,B,MAX,IERR)
```

and in C :

```
request = netslnb('dgesv()',n,1,a,max,ipiv,b,max,&ierr)
```

In case of an error, the request handler actually contains the (negative) NetSolve error code.
The next step is to check the status of the request. As in the Matlab interface, the user can chose to probe or to wait for the request. Probing is done by calling `netslpr()`which returns a NetSolve error code:

```
      call FNETSLPR(REQUEST,INFO)
```

and in C :

```
info = netslpr(request);
```

Typical error codes returned are `NetSolveNotReady` and `NetSolveOK` (see Section D). Waiting is done by using `netslwt()`. This function blocks until the computation is complete and the result is available. Here is the Fortran call:

```
      call FNETSLWT(REQUEST,INFO)
```

and the C call :

```
info = netslwt(request);
```

If the call is successful, the function returns the error code `NetSolveOK` and the result is in the user memory space.

4.3.5 Catching errors

There is a function in the C and Fortran interface that prints out explicit error messages to the standard error, given a NetSolve error code. The C call is :

```
netslerr(info);
```

and in Fortran

```
      call FNETSLERR(INFO)
```

4.3.6 Row- or column-major

To allow the NetSolve user to store her/his matrices either in row-wise or column-wise fashions, we also provide the function `netslmajor()` in C and `FNETSLMAJOR()` in Fortran. This function can be called at any time in the user's program in C:

```
netslmajor("col");
netslmajor("row");
```

or in Fortran:

```
call FNETSLMAJOR('col');
call FNETSLMAJOR('row');
```

All the subsequent calls to NetSolve will assume the corresponding major. The default values are of course row-wise for C and column-wise for Fortran.

4.3.7 Limitations of the Fortran interface

Due to to Fortran's restrictions for the use of pointer and its inability to dynamically allocate memory, the Fortran interface interface to NetSolve does not support the `PACKEDFILES` and `STRINGLIST` object type. It also does not support output objects of type `STRING`.

4.3.8 Built-in examples

C and Fortran and Java examples are included in the NetSolve distribution in `Netsolve-1.2.x/Examples`. To build them, the user simply types `make examples` in the top directory. The examples use different problems that have been given servers at the University of Tennessee. They should help the user to understand how the system works. We also have full examples in C and Fortran in Appendixes F and G.

4.4 Mathematica Interface

4.4.1 Introduction

Details on this interface can be found in [6] and instructions for building it are in the file:

`$NETSOLVE_ROOT/src/Mathematica/INSTALL`

4.4.2 What to do first

Once the interface is successfully installed, the first thing to do is to start a Mathematica client and type

```
   NetSolve[]
```

which prints out information on how to use the interface:

```
In[1]:= NetSolve[]
usage:
  NetSolve[FuncName[arg1, ...]]   - blocking problem call
  NetSolveNB[FUNCNAME[arg1, ...]] - nonblocking problem call
  NetSolveProbe[request]          - checks if a request is completed
  NetSolveWait[request]           - waits for a request to complete
  NetSolveGetAgent[]              - returns the current agent name
  NetSolveSetAgent[AgentName]     - changes the agent we are working with
  NetSolveError[]                 - returns the result code of the last
                                      executed NetSolve function
  NetSolveErrorMsg[rc]            - returns a string describing
                                      the result code passed
  NetSolve["?problems"]         - shows a list of available problems
  NetSolve["?servers"]          - shows a list of available servers
  NetSolve["?FuncName[]"]       - shows a problem description
```

Let us review the possibilities:

- Information functions
  `NetSolve["?problems"]`, `NetSolve["?servers"]` and `NetSolve["?FuncName[]"]`

- Blocking problem solving
  `NetSolve[ProblemName[arguments, ... ]]`

  Invocations of this type call NetSolve to solve a certain problem. When the result become available it is returned to the user. If the result is more than one object it is returned in a list.

- Nonblocking problem solving
  `NetSolveNB[ProblemName[arguments, ... ]]`

  The difference between blocking and nonblocking calls is that the later unlike the former the execution control is returned immediately to the user as well as a request handler. The last is used for checking if the result is available or the calculation is still in progress.

- Getting/setting an agent
  `NetSolveGetAgent[]`, `NetSolveSetAgent[AgentName]`

  `NetsolveGetAgent[]` returns back a string containing the host name of the agent. The user can change the current agent by `NetSolveSetAgent[]` function at any time.

  This set of functions is very important because they provide information about the calling sequence and what problems and servers are available trough the user's agent.

Let us now assume that the user has started Mathematica and is ready to try NetSolve. We can check what our agent is by typing

```
In[1]:= NetSolveGetAgent[]

Out[1]= unidhp.uni-c.dk
```

If there is no agent set, the result would be the `$Null` symbol. One can change the agent by the function `NetSolveSetAgent[]`. For instance

```
In[2]:= NetSolveSetAgent["bluegrass.cs.utk.edu"]
```

The agent can be changed at any time providing there is running another NetSolve agent on the host which name has been passed as argument. By doing this we actually change the set of servers and probably the set of solvable problems.
A list of the solvable problems can be obtained by the function `NetSolve["?problems"]`. Here is a possible list (clipped to save space).

```
In[3]:= NetSolve["?problems"]
/BLAS/Matrices/dgemm
/BLAS/Matrices/dmatmul
/BLAS/Matrices/zgemm
/BLAS/Matrices/zmatmul
/BLAS/Vectors/daxpy
/BLAS/Vectors/ddot
/BLAS/Vectors/zaxpy
/LaPack/Matrices/EigenValues/eig
/LaPack/Matrices/LinearSystem/dgesv
/LaPack/Matrices/LinearSystem/linsol
/MinPack/hybrd1
/MinPack/lmdif1
/QuickSort/DoublePrecision/dqsort
/QuickSort/Integer/iqsort
. . .
```

Similarly, a list of the servers can be printed by the function `NetSolve["?servers"]`

```
In[4]:= NetSolve["?servers"]
bluegrass.cs.utk.edu (128.169.92.41)
NetSolve Agent
Host: Up  Server: Running
cetus1a.cs.utk.edu (128.169.94.21)
Handles 24 problems
Host: Up  Server: Running
cetus1b.cs.utk.edu (128.169.94.22)
Handles 24 problems
Host: Up  Server: Running
cetus1c.cs.utk.edu (128.169.94.23)
Handles 24 problems
Host: Up  Server: Running
. . .
```

For every server or agent are printed its name, host and server status, IP address and how many different problems it can solve (for servers only).
The user can easily find out information about a problem, `iqsort` for instance, by typing

```
NetSolve["?iqsort[]"]
```

Please note the brackets after the problem name. They are necessary because every NetSolve problem is treated as a function defined in Mathematica. The output of that command can be divided to several parts. Let us look at them one by one.
The first part is a brief general description of the problem:

```
In[5]:= NetSolve["?iqsort[]"]
iqsort: Quicksort -
Sorts a vector of integers
```

The second part describes the input and output objects, their type and description. The input objects are what the problem requires as input, while the output objects are the result it produces:

```
Input:
  # 0 : Integer Vector
  Vector of integers to Sort

Output:
  # 0 : Integer Vector
  Sorted Vector
```

The last parts is intended to ease the user by providing him an example. The arguments showed in the example are variables but the user may wish to pass numerical values.

```
Mathematica example:
  rI0  = NetSolve[iqsort[I0]]

examples for types:

         Char     Byte/Integer  Single/Double  Complex
Scalar: "c"           42           66.32       4 - 7 I
Vector: "vector"    {1,2,3}     {3,4.5,7}     {3, -5+3I, 8}
Matrix: {"line 1", {{1,2,3},   {{6.4,2,1},    {{1+2I, 3+4I},
         "line 2"}  {4,5,6}}    {-7,1.2,4}}    {5-6I, 7}}
```

The number, the type and the sequence of arguments is important. If the user does not provide them in a proper way then the result would be an error message stating that the number of arguments is wrong or that a certain argument is not of a correct type.
When `NetSolve[]` is called to solve a particular problem an argument must be provided for every input object as well as for every output object of type file. The reason for the last is that NetSolve needs to know where to place the result.
The type and the dimension of the arguments should conform to the problem definition, otherwise an error message will be printed and the `$Null` symbol will be returned. The user can pass numerical values, symbols with assigned data to them or function calls. All of them must have proper type.
Here are some rules the user must take into account.

1. Characters are passed as strings (only the first character is used).
2. Integers can be passed instead of reals and vice versa (conversion is performed automatically).
3. Integers and reals can be passed instead of complexes.
4. Vectors of characters are passed as strings.
5. Matrixex of characters are passed as vectors of strings.

4.4.3 Blocking call to NetSolve

In the previous section we explained how the user can get information about a problem and its calling sequence. For the call itself, the problem name is passed together with its arguments (just like a function defined in Mathematica) to the function NetSolve which performs the computation. Let us resume our example:

```
In[6]:= NetSolve[iqsort[{7,2,3,5,1}]]
contacting server merlin.comlab ...

Out[6]= {1, 2, 3, 5, 7}
```

As stated earlier the user can pass not only numerical values, but also symbols that contain data of proper type or functions that return a result of this type. Indeed, Mathematica calculates these expressions and passes the arguments by value. For example

```
In[7]:= v = -Range[5]

Out[7]= {-1, -2, -3, -4, -5}

In[8]:= NetSolve[iqsort[v]]
contacting server merlin.comlab ...

Out[8]= {-5, -4, -3, -2, -1}
```

or to sort a random vector of size 7

```
In[9]:= NetSolve[iqsort[Table[Ceiling[10*Random[]], {7}]]]
contacting server merlin.comlab ...

Out[9]= {1, 2, 2, 2, 4, 6, 7}
```

`NetSolve[]` is a function defined in Mathematica. That is why it can be used in expressions like this

```
In[9]:= NetSolve[iqsort[Table[Ceiling[10*Random[]], {7}]]]
contacting server merlin.comlab ...

Out[9]= {1, 2, 2, 2, 4, 6, 7}

In[10]:= Print["The minimal element of v is ", NetSolve[iqsort[v]][[1]]]
contacting server merlin.comlab ...
The minimal element of v is -5
```

Let us try a more complex problem. For instance `dgemm[]`, which is a BLAS level 3 subroutine and calculates $\alpha \;.\; op(A) \;.\; op(B) + \beta \;.\; C$,
where op(X) = X or op(X) = X'
`dgemm[]` requires the following 7 arguments.

1. Character Scalar - specifies the form of op(A) to be used in the matrix multiplication as follows:
   'N' or 'n', op(A) = A
   'T' or 't', op(A) = A'
   'C' or 'c', op(A) = A'

2. Character Scalar - specifies the form of op(B) to be used in the matrix multiplication (see the previous argument)

3. Double precision real Scalar - α

4. Double precision real Matrix - Matrix A

5. Double precision real Matrix - Matrix B
6. Double precision real Scalar - β
7. Double precision real Matrix - Matrix C

Let us generate three random matrices.

```
In[11]:= RandomMatrix[m_,n_] := Table[Ceiling[10*Random[]], {m}, {n}]

In[12]:= a = RandomMatrix[2,3]

Out[12]= {{9, 2, 3}, {6, 3, 9}}

In[13]:= b = RandomMatrix[3,2]

Out[13]= {{6, 4}, {4, 10}, {2, 9}}

In[14]:= c = RandomMatrix[2,2]

Out[14]= {{4, 7}, {4, 8}}
```

and call `dgemm[]`.

```
In[15]:= NetSolve[dgemm["N", "N", 2, a, b, 3, c]]
contacting server cetus2a.cs.utk.edu ...

Out[15]= {{148., 187.}, {144., 294.}}

In[16]:= 2 a . b + 3 c

Out[16]= {{148, 187}, {144, 294}}
```

4.4.4 Nonblocking Call to NetSolve

As the Matlab interface (see Section 4.2), the Mathematica can be called in an asynchronous fashion. Nonblocking calls are performed by the function `NetSolveNB[]`. Its calling sequence is absolutely the same as blocking `NetSolve[]`. The difference is the result returned.`NetSolveNB[]` always returns back a request handler.
`NetSolveProbe[]` returns an integer value that shows if the problem has been solved. A value of 0 means that the result is available and 1 if not. Other values are error code (see Section 4.4.5).
Let us multiply two complex matrices using `NetSolveNB[]`. We generate the matrices `ac` and `bc` using already generated matrices `a`, `b` and `c`.

```
In[17]:= ac = a - 2 a I

Out[17]= {{9 - 18 I, 2 - 4 I, 3 - 6 I}, {6 - 12 I, 3 - 6 I, 9 - 18 I}}

In[18]:= bc = b - 3 b I

Out[18]= {{6 - 18 I, 4 - 12 I}, {4 - 12 I, 10 - 30 I}, {2 - 6 I, 9 - 27 I}}

In[19]:= request = NetSolve[zmatmul[ac, bc]]
```

```
contacting server cetus2a.cs.utk.edu ...

Out[19]= 0

In[20]:= NetSolveProbe[request]

Out[20]= 0
```

Not ready yet. We can do some other work ...

```
In[21]:= NetSolveWait[request]

Out[21]= {{-340. - 340. I, -415. - 415. I}, {-330. - 330. I, -675. - 675. I}}
```

4.4.5 Catching Errors

Similarly to the Matlab interface, it is possible to detect errors with the functions `NetSolveError[]` and `NetSolveErrorMsg[]`. The first returns an integer which is the error code of the last executed NetSolve function. `NetSolveErrorMsg[]` takes an error code as argument and returns a string describing the error. When an error occurs a message is printed on the user's display. In this case the user do not need the error functions. They are only useful when the user would like to write a fault tolerant Mathematica script.

4.5 Java API

The Java API is being converted to NetSolve 1.2.

4.6 Java GUI

The Java GUI is being converted to NetSolve 1.2.

Chapter 5

NetSolve Request Farming

Farming is a new way of calling NetSolve to manage large numbers of requests for a single NetSolve problem. Many NetSolve users are confronted to situations when many somewhat similar computations must be performed in parallel. The way to do this in NetSolve is to write calls to `netslnb()` in C for instance. However, this becomes very cumbersome. Not only because the user has to manage all the requests himself, but also because the NetSolve system is at a loss trying to manage such a large number of requests without flooding the servers. This is the motivation for distributing a new call in NetSolve: `netsl_farm()`. In the present distribution, this call is only available from C, but will soon be made available from Matlab, Mathematica, and Java. Fortran will most likely be left out because of pointer management. For now, linking to the C NetSolve client library (generated as explained in Section 3.4) makes `netsl_farm()` available from the user's program.

5.1 How to call farming

Like `netsl()` and `netslnb()`, the `netsl_farm()` function takes a variable number of arguments. Its first argument is a string that describes the *iteration range*. This string is of the form `"i=%d,%d"` (in C string format symbols). The second argument is a problem name postpended with an opening and a closing parenthesis. The arguments followings are similar in intent to the ones supplied to `netsl()`, but are *iterators* as opposed to integers or pointer. Where the user was passing, say an integer, to `netsl()`, he now needs to pass an array of integers and tell `netsl_farm()` which element of this array is to be used for which iteration. This information is encapsulated in an *iterator* and we provide three functions to generate iterators:

- `ns_int()`
- `ns_int_array()`
- `ns_ptr_array()`

Let us review these functions one by one.

5.1.1 `ns_int()`

This function takes only one argument: a character string that contains an *expression* that is evaluated to an integer at each iteration. The format of that string is based on a Shell syntax. `$i` represents the current iteration index, and classic arithmetic operators are allowed. For instance:

```
ns_int("$i+1")
```

returns an iterator that generates an integer equal to one plus the current iteration index at each iteration.

5.1.2 ns_int_array()

This function takes two arguments: (i) a pointer to an integer array (`int *`); (ii) a character string that contains an expression. For instance,

```
ns_int_array(ptr,"$i")
```

returns an iterator that generates at each iteration an integer equal to the i-th element of the array `ptr` where i is the current iteration index.

5.1.3 ns_ptr_array()

This function takes two arguments: (i) a pointer to an array of pointers (`void **`); (ii) a character string that contains an expression. For instance,

```
ns_ptr_array(ptr,"$i")
```

returns an iterator that generates at each iteration a pointer which is the i-th element of the array `ptr` where i is the current iteration index.

5.2 An example

Let us assume that the user wants to sort an array of integers with NetSolve using the C interfaces. The default NetSolve server comes with a default problem called `iqsort` that does a quicksort on an integer vector. The call looks like

```
status = netsl('iqsort()',size,ptr,sorted);
```

where `size` is the size of the array to sort, `array` is a pointer to the first element of the array, and `sorted` is a pointer to the memory space that will hold the sorted array on return. What if the user wants to sort 200 arrays? On way is to write 200 calls as the one above. Not only would it be tedious, but also inefficient as the sorts would be done successively, with no parallelism. In order to obtain some parallelism, one must call `netslnb()` and make the corresponding calls to `netslpr()` and `netslwt()` as explained in Section 4.3. Again, this is pretty tedious and as it is a pretty common situation we decided to address it with `netsl_farm()`. Before calling `netsl_farm()`, the user needs to construct arrays of pointers and integers that contain the arguments of each of the NetSolve call. This is pretty straightforward: where the user would have called NetSolve as:

```
requests1 = netslnb('iqsort',size1,ptr1,sorted1);
requests2 = netslnb('iqsort',size2,ptr2,sorted2);
...
requests200 = netslnb('iqsort',size200,array200,sorted200);
```

and then to have calls to `netslpr()` and `netslwt()` for each request.
With farming, one just needs to construct three arrays as:

```
int size_array[200];
void *ptr_array[200];
void *sorted_array[200];

size_array[0] = size1;
ptr_array[0] = ptr1;
sorted_array[0] = sorted1;
...
```

Then, `netsl_farm()` can be called as:

```
status_array = netsl_farm("i=0,199",netsl_int_array(size_array,"$i"),
                                    netsl_ptr_array(ptr_array,"$i"),
                                    netsl_ptr_array(sorted_array,"$i"));
```

In short, `netsl_farm()` is a concise, convenient way of farming out groups of requests. Of course, it uses `netslnb()` underneath, thereby ensuring fault-tolerance and load-balancing.

5.3 Catching errors

`netsl_farm()` returns an integer array. That array is dynamically allocated and must be freed by the user after the call. The array is at least of size 1. If the first element of the array is either 0 or -1. If it is 0, then the call was completed successfully and the array is of size 1. If first element of the array is -1, then at least one requests failed. The array is then of size one plus the number of requests and the (1+i)-th element of the array is the error code for the i-th request. Here is an example on how to print out error messages:

```
status = netsl_farm("i=0,200",....);
if (status[0] == 0){
   fprintf(stderr,"Success\n");
   free(status);
} else {
  for (i=1;i<201;i++) {
    fprintf(stderr,"Request #%d:",i);
    netslerr(status[i]);
  }
}
free(status);
```

5.4 Current Implementation and Future Improvements

One of the advantages of farming is that the user does not have the responsibility to handle the management of requests. For instance, it would be unreasonable to send all the requests if there are not enough servers to perform the computations. Instead, `netsl_farm()`'s implementation takes care of that. Without going into detail, let us just say that the farming algorithm dynamically tunes the maximum number of pending requests to reflect changes in the computational server pool (size and load). This is done by constantly measuring the throughput of the computations.

Chapter 6

The NetSolve Agent and Server

6.1 The Agent

After compiling the agent as explained in Section 3.4, the executable of the NetSolve agent is located in:

`$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/agent.`

This executable can be called with no argument as

```
% agent &
```

and this starts a stand-alone agent. This agent will be available for NetSolve servers to participate in a new NetSolve system. The executable can also take one single argument that is the name of a host already running a NetSolve agent:

```
% agent bluegrass.cs.utk.edu
```

for instance. This starts a NetSolve agent on the local hosts and connects it to an existing NetSolve system that can consist of multiple agents (and servers). The local agent becomes then a new client entry-point to that system.

6.2 The Server

6.2.1 Starting a Server

After compiling the server as explained in section 3.4, the executable of the NetSolve server is located in:

`$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/server.`

This executable uses a *configuration file* for initializing the NetSolve server. It can be called with no argument as:

```
% server &
```

in which case the default configuration file located in `$NETSOLVE_ROOT/server_config` is used. This is the file that should be used for first experiments and for testing the system. However, it is possible to specify another configuration file by calling the executable as:

```
% server /home/me/my_config &
```

for instance. In the following section, we explain the structure of the server configuration file.

6.2.2 The Server Configuration File

The configuration file is organized in lines. A line can start with a `'#'` in which case the line is ignored and can be used for comments. A line can also start with a *keyword* that is prefixed by a `'@'`. Such a line is said to start a *section* of the configuration file. A section can consist of only the line with the keyword. Let us review all the possible keywords and how they can be used to precisely define a NetSolve server as it is done in the default configuration file.

- `'@AGENT:<hostname>'[*]` specifies the agent that the NetSolve server must contact to register into a NetSolve system. The agent is identified by the name of the host on which it is running and there can be only one such line in the configuration file. If the '*' is present, then the server will broadcast its existence to all NetSolve agents known to the one running on `<hostname>`. Otherwise, the server will only be known to the agent on `<hostname>`.

- `'@PROCS:<number>'` specifies the number of processors that can be used by the server to perform simultaneous computations on the local hosts. There can only be one such line in the configuration file.

- `'@MPIHOSTS <filename>'` specifies the path to the file that contains the list of machines that can be used by MPI.

- `'@MPIPATH <path>'` specifies the path to the directory that contains the MPI executables (e.g. `mpirun`).

- `'@WORKLOADMAX:<max>'` specifies the value of the workload beyond which the server refuses new requests (e.g. `'@WORKLOADMAX:100'`). A value of `-1` means that the server accepts requests regardless of the workload.

- `'@SCRATCH:<path>'` specifies where the NetSolve server can put temporary directories and file. The default is `/tmp/`.

- `'@CONDOR:<path>'` specifies that the NetSolve server is using a Condor [7, 8] pool as a computing resource. The path to the Condor base directory must be provided. There can be only one such line in the configuration file.

- `'@PROBLEMS:'` marks the beginning of the list of *description file* names. The problems from these description files must be added to the server. Details on description files are given in Section 6.3.

- `'@RESTRICTIONS:'` marks the beginning of the list of access restrictions that are applicable to the NetSolve server. The list consists of lines formatted as:

  ```
  <domain name>  <number of pending requests allowed>
  ```

  The symbol `'*'` is used as a wildcard in the domain name. For instance, the line:

  ```
  *.edu 10
  ```

  means that only 10 requests from clients residing on a `.edu` machine can be serviced simultaneously. When the server receives a request from some machine, it determines which line in the list must be used to accept or reject the request by taking the most refined domain name. For instance, if the list of the restrictions is:

  ```
  *.edu 5
  *.utk.edu 10
  ```

then the server accepts at most 5 simultaneous requests coming from `.edu` machines that are **not** in the *.utk.edu* sub-domain, and at most 10 requests that come from machines in the `.utk.edu` sub-domain for a total of 15 possible simultaneous requests.

The default configuration file in `$NETSOLVE_ROOT/server_config` should be used as a template to create new configuration files.

6.2.3 Customizing the Server

The default server configuration file contains several lines that specify description files. These files are located in the `$NETSOLVE_ROOT/problems` directory. This directory contains many description files that are not used by the default NetSolve server. These files correspond to problems that are solved with numerical software that is not distributed with NetSolve but can be added by uncommenting the corresponding lines in the server description file. The server can then be recompiled by typing

```
% make server
```

in the `$NETSOLVE_ROOT` directory. Section 6.3 gives details on the creation of new problem description files.

6.3 Expanding the Server

As already indicated in Section 6.2.3, it is possible to add new functionalities to a NetSolve computation server by specifying additional description files in the server configuration file. In what follows we describe how a description file can be created. It is strongly advised to use the existing files as templates. At the moment this document is being written, the NetSolve development team is testing a Java applet that should make the generation of problem description files easy and interactive. When this applet reaches a mature enough state, it will be put on the NetSolve homepage.
The rationale behind everything that is explained in what follows comes from [9]. Each description file is composed of several *problem descriptions*. Before explaining how to create a problem description, we define the concept of *mnemonics*.

6.3.1 Mnemonics

A NetSolve problem takes some objects in input and produces some objects as output. To be able to relate high-level and low-level descriptions of the input and output objects of a given problem, we need to develop some kind of syntax. We decided to call the member of this syntax *mnemonics*. A mnemonic is just a character string (typically 2 or 3 characters long) that is used to access low level details of the different input and output objects. We index the list of objects, starting at 0. Therefore, the first object in input to a problem is the input object number 0 and the third object in output to a problem is the output object number 2, for instance. We use an `I` or an `O` to specify whether an object is in input or output. Here are the four types of mnemonics for an object indexed x:

- Pointer to the data : `[I|O]`x,
- Number of rows : `m[I|O]`x (only for matrices, vectors, packed files and string lists),
- Number of columns : `n[I|O]`x (only for matrices),
- Leading dimensions : `l[I|O]`x (only for matrices).
- Special descriptor : `d[I|O]`x (only for distributed memory objects).

For example, `mI4` designates the number of rows of the input object number 4, whereas `O1` designates the pointer to the data of output object number 1. In the next section, we describe the different sections that are necessary to build a problem description and will see how the mnemonics are used.

6.3.2 Sections of a Problem Description

The structure of a problem description file is very similar to the one of a server configuration file. The lines starting with a `'#'` are considered comments. Keywords are prefixed by a `'@'` and mark the beginning of sub-sections. In what follows, we describe each section separately as well as each keyword and sub-sections within each section. Keep in mind to look at one existing problem description file as a template when reading this section.

Problem ID and General Information

- `'@PROBLEM <nickname>'` specifies the name of a problem as it will be visible to the NetSolve users (clients).
- `'@INCLUDE <name>'` specifies a C header file to include (See the example in Section 6.3.4). There can be several such lines as a problem can call several functions.
- `'@DASHI <path>'` specifies a default directory in which header files are to be looked for, in a similar way as the `-I` option of most C compilers. There can be several such lines as a problem can call several functions.
- `'@LIB <name>'` specifies a library or an object file to link to, or a $-L$ option for the linker (See the example in Section 6.3.4). There can be several such lines and the lines can contain variable substitutions such as `$(NETSOLVE_ROOT)`.
- `'@FUNCTION <name>'` specifies the name of a function from the underlying numerical software library that is being called to solve the problem. There can be several such lines as a problem can call several functions.
- `'@LANGUAGE [C|FORTRAN]'` specifies whether the underlying numerical library is written in C or in Fortran. This is used in conjunction with the function names specified with `'@FUNCTION'` to handle multi-language interoperability.
- `'@MAJOR [COL|ROW]'` specifies what major should be used to store the input matrices before calling the underlying numerical software. For instance, if the numerical library is LAPACK [10], the major has to be `'COL'`.
- `'@PATH <path>'` specifies a path-like name for the problems. This path is only a naming convention and is used for presentation purposes.
- `'@DESCRIPTION'` marks the beginning of the textual description of the problem. This sub-section is mandatory as it allows NetSolve users (clients) to find out what a problem can do.

Input Specification

- `'@INPUT <number>'` specifies the number of objects in input to the problem. This line is followed by a corresponding number of object descriptions (see below).
- `'@OBJECT <object type> <data type> <name>'` specifies an object type, data type, and name. The name is really only for presentation purposes. This line is followed by a mandatory textual description of the object. The types are abbreviated by replacing `NETSOLVE_X` by `X` (see table 2.1). The objects of object type `FILE`, `STRING`, `UPF`, and `PACKEDFILES` do not have a data type. Here are a few examples:

  ```
  @OBJECT VECTOR I X
  An integer vector named 'X'
  ```

```
@OBJECT MATRIX D A
A double precision real matrix named 'A'

@OBJECT FILE foo
A file named 'foo'
```

Output Specification

- '@OUTPUT <number>' specifies the number of objects in output to the problem. This line is followed by a corresponding number of object descriptions (see below).
- '@OBJECT <object type> <data type> <name>' specifies an object type, a data type and a name. This line is followed by a mandatory textual description of the object.

Additional Information

- '@MATLAB_MERGE <number1>,<number2>' specifies that the output objects number $< number1 >$ and $< number2 >$ can be merged as a complex object upon reception of the numerical results from the Matlab client interface (see Section 4.2.2).
- '@COMPLEXITY <number1>,<number2>' specifies that given the size of the problem, say n, the asymptotic complexity, say C, of the problem in number of floating point operations is

$$C = number1 \times n^{number2}.$$

6.3.3 Calling Sequence

The calling sequence to the problem must be defined here so that the NetSolve client using the C or Fortran interfaces can call the problem. The material described in this section is ignored by NetSolve when the client is Matlab, Mathematica or Java. To clarify, let us take an example. Let us say that the problem *'toto'* takes a matrix in input and returns a matrix in output. The call from the Matlab interface looks like:

```
>> [b] = netsolve('toto',a)
```

for instance. However, there can be several possible calling sequences from C or Fortran. Assuming the following declarations in Fortran:

```
DOUBLE PRECISION A(M,N)
        DOUBLE PRECISION B(K,L)
```

the following calling sequences are all possible:

```
CALL FNETSL('toto()',A,B,M,N,K,L)
CALL FNETSL('toto()',A,M,N,B,K,L)
CALL FNETSL('toto()',M,N,A,K,L,B)
        etc.....
```

The Calling sequence sub-section in the problem description is used to specify which calling sequence is to be used thanks to mnemonics. Indeed, still with the same example, the integer `N` can be represented by the mnemonic `nI0`, and the pointer B can be represented by the mnemonic `O0`.

- '@CALLINGSEQUENCE' marks the beginning of a calling sequence description. This description consists of a list of argument specifications (see below).
- '@ARG <comma-separated list of mnemonics>' specifies an argument of the calling sequence. For instance the line

```
@ARG I0
```

specifies that the current argument in the calling sequence is the pointer to the data of the first object in input. The line

```
@ARG mI0,lI0
```

specifies that the current argument in the calling sequence is the number of rows **and** the leading dimension of the first object in input (which in this case is a matrix). The line

```
@ARG ?
```

specifies that the current argument in the calling sequence should be ignored by NetSolve (useful in some cases). Note that no argument description contains mnemonics of the form `[m|n]O*`.

- `'@CONST <mnemonic>=<number>'` specifies that the number of rows or columns or the leading dimension of an input object is constant and can not be found in the calling sequence. For instance, the line

  ```
  @CONST mI4=12
  ```

  means that the number of rows of the fifth object in input is always 12 and is not passed in by the NetSolve user.

- `'@COMP <mnemonic>=<expression>'` specifies that the number of rows or columns or the leading dimension of an input object can be computed from the calling sequence but can not be found in the calling sequence. Here are some examples:

  ```
  @COMP mI1=mI0
  @COMP mI0=op(+,mI3,1)    // performs an addition
  @COMP mI3=array(I2,0)    // performs an indirection
  @COMP mI1=op(-,array(I0,op(-,mI0,1)),1)
  @COMP mI2=op(+,op(+,array(I1,0),1),op(*,array(I0,0),2))
  @COMP mI2=if(array(I0,0)='N',mI1,if(array(I0,0)='T',nI1,op(-,0,1)))
                    // conditionals
  ```

  Things can get quite complex. However, our experience proves that this feature is used only rarely. Therefore, we have not yet concentrated our efforts on making this particular process easier. Especially since the aforementioned Java applet will make the creation of the problem description files automatic.

Pseudo-Code

- `'@CODE'` marks the beginning of the pseudo-code section.
- `'@END_CODE'` marks the end of the pseudo-code section.

The pseudo-code is C code that uses the mnemonics described in Section 6.3.1. This code contains call(s) to the numerical library function(s) that the problem is supposed to use as part of its algorithm. The arguments in the calling sequences of these library routines will be primarily the different mnemonics. In the pseudo-code, the mnemonics are pre- and post-fixed by a `'@'` to facilitate the parsing. Let us review again the meaning of some possible mnemonics in the pseudo-code:

- `'@I0@'`: pointer to the elements of the first object in input.
- `'@mI0@'`: **pointer** to an integer that is number of rows of the first object in input.

- '@nO1@': **pointer** to an integer that is number of columns of the second object in output.

Usually, the pseudo-code is organized in three parts. First, the *preparation* of the input (if necessary). Second, the call the numerical library function(s). Third, the update of the output (pointer and sizes). At this point, it is best to give an example. Let us assume that we have access to a hypothetical numerical C library that possesses a function `matvec()` that performs a matrix-vector multiply for square matrices. The prototype of the function is

```
void matvec(float *a, float *b, int n, int l),
```

where `a` is a pointer to the matrix, `b` is a pointer to the vector, `n` is the dimension of the matrix, `l` is the leading dimension of the matrix and the result is stored in `b` (overwriting the input). We may define the problem such that the matrix is the first object in the input, the vector the second object in the input, and the result the only object in output. Possible preparations could be for instance the creation of workspace, test of input values to detect mistakes, test of matching dimensions. In this case, we may want to check that the dimension of vector `b` agrees with the number of columns of matrix `a`. This can be done as follows:

```
@CODE
if (*@mI1@ != *@nI0@)
  return NS_PROT_DIM_MISMATCH;
```

The macro `NS_PROT_DIM_MISMATCH` is defined by NetSolve. Other macros available are `NS_PROT_BAD_VALUES` (for invalid input parameters), `NS_PROT_INTERNAL_FAILURE` (for a malfunction of the numerical software) or `NS_PROT_NO_SOLUTION` (sometimes useful if no numerical solution has been found and the client is interactive). Notice the use of `'*'` for accessing the integers at addresses `@mI1@` and `@nI0@`.
The second part of the pseudo-code consists in calling the function `matvec` and is:

```
matvec(@I0@,@I1@,*@mI0@,*@mI0@);
```

A few things can be said on this call. First, we use the `'*'` to access integers via the pointers. Note that if `matvec()` were a Fortran subroutine, we would pass the addresses themselves (see Example below). Second, the leading dimension is taken to be equal to the dimension. This code is executed at the server level where the matrix (or sub-matrix) has been received from the client over the network. As such, it has been stored contiguously in memory and has a leading dimension equal to its number of rows. As a general rule, the mnemonics `@l[I|O]*@` never appear in the pseudo-code. The last thing to do at this point is to update the output:

```
@O0@ = @I1@;
*@mO0@ = *@mI1@;
@END_CODE
```

The first line expresses the fact that the input has been overwritten by the output. The second line sets the number of rows of the output. The following section gives a complete example, with all the sections of the problem description.

6.3.4 A Simple Example

Let us imagine that we have access to a Fortran numerical library that contains a function, say `LINSOL`, to solve a linear system according to the following prototype:

```
SUBROUTINE LINSOL(A,B,N,NRHS,LDA,LDB)

DOUBLE PRECISION A(LDA,*)  // Left-hand side (NxN)
DOUBLE PRECISION B(LDB,*)  // Right-hand side (NxNRHS),
```

```
                           // overwritten with the solution
INTEGER N
INTEGER NRHS
INTEGER LDA               // Leading Dimension of A
INTEGER LDB               // Leading Dimension of B
```

Then, an appropriate description for a problem that solves a linear system using `LINSOL` and that expects from the client the same calling sequence as the one for `LINSOL` is:

```
@PROBLEM linsol
@INCLUDE <math.h>
@INCLUDE "/home/me/my_header.h"
@LIB -L/home/lib/
@LIB -lstuff
@LIB /home/me/lib_$(NETSOLVE_ARCH).a
@LIB /home/stuff/add.o
@FUNCTION linsol
@LANGUAGE FORTRAN
@MAJOR COL
@PATH    LinearAlgebra/LinearSystems/
@DESCRIPTION
Solves the square linear system A*X = B. Where:
 A is a double-precision matrix of dimension NxN
 B is a double-precision matrix of dimension NxNRHS
 X is the solution
@INPUT 2
@OBJECT MATRIX D A
Matrix A (NxN)
@OBJECT MATRIX D B
Matrix B (NxNRHS)
@OUTPUT 1
@OBJECT MATRIX D X
Solution X (NxNRHS)
@COMPLEXITY 3,3
@CALLINGSEQUENCE
@ARG I0
@ARG I1,O0
@ARG nI0,mI0,mI1
@ARG nI1
@ARG lI0
@ARG lI1,lO0
@CODE

linsol(@I0@,@I1@,@mI0@,@nI1@,@lI0@,@lI1@);

@O0@ =@I1@;       /* Pointing to the overwritten input */
*@mO0@ = *@mI1@;  /* Setting the number of rows        */
*@nO0@ = *@nI1@;  /* Setting the number of columns     */

@END_CODE
```

6.3.5 Java Applet

It appears that the process of creating new problem descriptions can be very difficult, especially for a first time user. It is true that after writing a few files, it becomes rather routine and several NetSolve users have already generated a good number of working description files for a variety of purposes (including Linear Algebra, Optimization, Image processing, etc.). However, we have designed a graphical Java Applet that helps users in creating problem description files. This applet is not yet available from the Web at the time this document is being written but should be added to the NetSolve homepage as soon as the last tests and conversion to Java1.1 have been completed.

6.4 NetSolve Management Tools

The NetSolve distribution comes with a set of rudimentary tools to manage/query a NetSolve system. After compiling the tools as explained in Section 3.4, the following six executables are available:

- `$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/NS_conf`
- `$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/NS_problems`
- `$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/NS_probdesc`
- `$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/NS_killagent`
- `$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/NS_killserver`
- `$NETSOLVE_ROOT/bin/$NETSOLVE_ARCH/NS_killall`

Let us review those executables one by one.

6.4.1 NS_conf

This executable takes one argument on the command line, the name of a host running a NetSolve agent:

```
% NS_conf bluegrass.cs.utk.edu
```

It prints out the list of hosts participating in the NetSolve system:

```
AGENT: bluegrass.cs.utk.edu (128.169.92.78)
SERVER: maruti.cs.berkeley.edu (128.32.36.83)
SERVER: cupid.cs.utk.edu (128.169.94.221)
```

6.4.2 NS_problems

This executable takes the name of a host running an agent as single argument on its command line. It prints out the list of problems that can be solved by contacting that agent:

```
% NS_problems bluegrass.cs.utk.edu
/BLAS/Matrices/matmul
/ItPack/jsi
/LaPack/Matrices/EigenValues/eig
/LaPack/Matrices/SingularValues/svd
```

6.4.3 NS_probdesc

This executable takes two arguments on its command line: the name of a host running a NetSolve agent and the nick name of a NetSolve problem. It prints out the description of the problem:

```
% NS_probdesc bluegrass.cs.utk.edu linsol
-- linsol -- From LAPACK -
Compute the solution to a real system of linear equations
  A * X = b
where A is an N-by-B matrix and X and B are N-by-NRHS matrices.
Matlab Example : [x] = netsolve('dgesv',a,b)
http://www.netlib.org/lapack/index.html
* 2 objects in INPUT
 - input 0: Matrix Double Precision Real.
 Matrix A
 - input 1: Matrix Double Precision Real.
 Right hand side
* 1 objects in OUTPUT
 - output 0: Matrix Double Precision Real.
 Solution
* Calling sequence from C or Fortran
6 arguments
 - Argument #0:
   - number of rows of input object #0 (A)
   - number of columbs of input object #0 (A)
   - number of rows of input object #1 (RHS)
 - Argument #1:
   - number of columbs of input object #1 (RHS)
 - Argument #2:
   - pointer to input object #0 (A)
 - Argument #3:
   - leading dimension of input object #0 (A)
 - Argument #4:
   - pointer to input object #1 (RHS)
   - pointer to output object #0 (SOLUTION)
 - Argument #5:
   - leading dimension of input object #1 (RHS)
```

6.4.4 NS_killagent

This executable takes one single argument on its command line, the name of a host running a NetSolve agent. After a (basic) user authentication, that executable kills the agent.

```
% NS_killagent bluegrass.cs.utk.edu
Agent on bluegrass.cs.utk.edu :  killed
```

6.4.5 NS_killserver

This executable takes two single argument on its command line, the name of a host running a NetSolve agent and the name of a host running a NetSolve server. After a (basic) user authentication, that executable kills the server, using the agent as an entry-point into the system.

```
% NS_killserver bluegrass.cs.utk.edu cupid.cs.utk.edu
```

```
Server on cupid.cs.utk.edu killed : killed
```

6.4.6 NS_killall

This Shell script takes one single argument on its command line, the name of a host running a NetSolve agent. After a (basic) user authentication, that executable kills the agent, along with all other NetSolve processes (agents and servers) known to that agent:

```
Server on cupid.cs.utk.edu : killed
Server on maruti.cs.berkeley.edu : killed
Agent on bluegrass.cs.utk.edu : killed
```

Chapter 7

The User-Supplied Function Feature

7.1 Motivation

In the preceding sections, we described all the client interfaces to NetSolve. In these descriptions we assumed that the only input the user had to supply to NetSolve was numerical data, that is, matrices, vectors, or scalars. This assumption is valid for a lot of numerical software. However, for some software that we would like to include in NetSolve via NetSolve servers, we need an additional feature. Indeed, numerous scientific packages require the user to provide numerical data as well as a *function*. Typically, nonlinear software requires the user to pass a pointer to a subroutine that computes the nonlinear function. This is a problem in NetSolve because the computation is performed remotely and the user cannot provide NetSolve with a pointer to one of his linked-in subroutines. The only solution is to send code over the network to the server. This approach raises a lot of issues, including *security*.

7.2 Solution

Let us describe here the solution we have adopted. This is really a first attempt, and there is definitely room for improvement. However, we believe that it provides reasonable capabilities for now, considering that NetSolve is still at an early stage of development. As we noted, we need to ship code over to the computational server. Since NetSolve works in a heterogeneous environment, it is not possible to migrate compiled code. Thus, we require that the user have his subroutine or function in a separate file, written either in C or Fortran. We send this file to the computational server. The server compiles it and is then able to use this user-supplied function.
The security implementation is quite simple. When compiling the user's function, we use the `nm` UNIX command to disallow any system call. The approach is very restrictive for the user, but typically the subroutine that has to be passed needs only to perform computations. If course, there are a lot of *hacker* ways to go around this problem, and our system currently does not pretend to be a real security manager. We are investigating Java to deal with this user-supplied function issue.

7.3 For the Client

7.3.1 Determining the Format of the Function to Supply

We now understand that the user has to write a Fortran subroutine or a C function to call a problem that requires a user-supplied function. For now, the prototype of this subroutine/function can be found in the description of the problem, available from Matlab or the CGI scripts of the NetSolve homepage (see

Section 2.3). Following the usual philosophy of NetSolve, the prototype of the user-supplied function is exactly the same as if the user were using the numerical software directly. Some softwares require the user to provide more than one function. When that is the case, the description of the problem mentions it and gives all the prototypes for all the functions to supply.

7.3.2 From Matlab, Mathematica, C and Fortran

A UPF is passed to NetSolve as a string that contains the path to the file that contains the source code of the function.

7.3.3 From the NetSolve Java API

Users of the NetSolve API may specify a UPF input item as they would any other input item, using the `pushArg()` method. However, an extra argument is required when pushing a UPF item: the language that the UPF is written in. For example:

```
n.pushArg(new String(upf0,0),GlobalDefs.LANG_FORTRAN);
n.pushArg(new String(upf1,0),GlobalDefs.LANG_C);
```

Currently, the user must pass the UPF as a String. Therefore, if the UPF is stored in a file, it is up to the user to read the file into a String. Future versions of the API will allow the user to simply pass the name of the file.

7.3.4 From the Java GUI

Entering a user-supplied function via the Java interface is very much similar to entering any other kind of data. If the problem requires a user-supplied function, there will be an entry in the *Input List* called "User Provided Function" for which data must be specified, just like any other input object. The user may choose to enter the user-supplied function manually into the *Data Input Box* or from a file specified in the *Filename Selection Box*. If the user enters the function manually, the language must also be specified by choosing either C or FORTRAN from an "option menu" that appears just above the *Data Input Box*. If the user-supplied function comes from a file, the file must end with either ".c" or ".f" (with names ending in ".c" interpreted as C functions and names ending in ".f" interpreted as FORTRAN functions).

7.4 For the Server

The problem description of a problem that requires one or more user-supplied functions must contain a line:

```
@OBJECT UPF CHAR
```

for each function as an input object so that mnemonics can be used in the description of the calling sequence (after the `'@FORMAT'` clause). In the pseudo-code section, the functions should be declared as extern like:

```
extern int upf0();
extern double upf1();
etc....
```

for instance. The identifiers `upf0`, `upf1`, ... can be used in the rest of the pseudo code to designate the user-supplied functions. This is not very natural. It would be better to be able to use mnemonics as for classic objects, but it makes compilation difficult on some platforms.

7.5 Conclusion

This new feature of NetSolve is still under investigation. We are aware that security is an important issue here. For now, NetSolve is still a research project developed to allow experimentations with this relatively new type of software. In the future, more attention will be given to the user-supplied mechanism in order to make it as safe as possible. As mentioned earlier, we may use Java in order to set up a viable security manager. Using Java currently appears to be the best solution for security, but it has obvious drawbacks. First, the user would have to write his function in Java: the typical NetSolve user is a scientist who does not have the time or inclination to learn new languages, especially object-oriented ones. Second, with the current implementations of Java, efficiency would also be a problem.

Appendix A

Matlab Reference Manual

We describe here all the possible calls to NetSolve from Matlab. In these descriptions we assume correctness. In case of errors, all these calls print out very simple and explicit messages.

>> netsolve

Prints out on the screen the list of all the problems that are available in the NetSolve system.

>> netsolve('<*problem name*>')

Prints out all the information available from Matlab about a specific problem.

>> netsolve('?')

Prints out the list of all the agents and servers in the NetSolve system, that is, the NetSolve system containing the host whose name is in the environment variable NETSOLVE_AGENT.

>> [...] = netsolve('<*problem name*>', ...)

Sends a **blocking** request to NetSolve. The left-hand side contains the output arguments. The right-hand side contains the problem name and the input arguments. The arguments are listed according to the problem description. Upon completion of this call, the output arguments contain the result of the computation.

>> [r] = netsolve_nb('send','<*problem name*>', ...)

Sends a **non-blocking** request to NetSolve. The right-hand side contains the keyword send, the problem name, and the list of input arguments. These arguments are listed according to the problem description. The left-hand side will contain a request handler upon completion of the call.

>> [...] = netsolve_nb('wait',r)

`Waits` for a requests completion. The right-hand side contains the keyword `wait` and the request handler. The left-hand side contains the output arguments. These arguments are listed according to the problem description. The right-hand side contains the keyword `wait` and the request handler. Upon completion of this call, the output arguments contain the result of the computation.

>> [status] = netsolve_nb('probe',r)

`Probes` for a request completion. The right-hand side contains the keyword `probe` and the request handler. The left-hand side contains the output arguments. These arguments are listed according to the problem description. The right-hand side contains the keyword `probe` and the request handler. Upon completion of this call, the output arguments contain the result of the computation.

>> netsolve_nb('status')

Prints out the list of all the pending requests. This list contains estimated time of completion, the computational servers handling the requests and the current status. The status can be `COMPLETED` or `RUNNING`.

>> nesolve_err

Returns the error code of the most recently called NetSolve function.

>> netsolve_errmsg(e)

Returns a string that contains he error message that corresponds to the error code passed as the argument.

Appendix B

C Reference Manual

We describe here all the possible calls to NetSolve from C. All these calls return a NetSolve code status. The list of the possible code status is given in Appendix D.

`status = netsl("`<*problem name()*>`()", ...)`

Sends a **blocking** request to NetSolve. `netsl()` takes as argument the name of the problem and the list of arguments in the calling sequence. See Section 4.3.2 for a discussion about this calling sequence. It returns the NetSolve status code (integer `status`). If the call is successful, the result of the computation is stored in the output arguments. The output arguments are specified in the calling sequence.

`status = netslnb("`<*problem name()*>`()", ...)`

Sends a **nonblocking** request to NetSolve. `netslnb()` takes as argument the name of the problem, and the list of arguments in the calling sequence. See Section 4.3.2 for a discussion about this calling sequence. It returns the NetSolve status code (integer `status`). If the call is successful, `status` contains the request handler.

`status = netslwt(`<*request handler*>`)`

Waits for a request completion. `netslwt()` takes as argument a request handler (an integer). If the call is successful, the result of the computation is stored in the output arguments. The output arguments are specified in the calling sequence during the call to `netslnb()`.

`status = netslpr(`<*request handler*>`)`

Probes for a request completion. `netslpr()` takes as argument a request handler (an integer). If the call is successful, the result of the computation is stored in the output arguments. The output arguments are specified in the calling sequence during the call to `netslnb()`.

`netslerr(`<*error code*>`)`

Displays an explicit error message given a NetSolve error code.

`netslmajor("<major>")`

Sets the way the user has stored her matrices (row- or column-wise). The argument can be `"col"` or `"row"`. It is case-insensitive and in fact only the first character is used by NetSolve.

Appendix C

Fortran Reference Manual

We describe here all the possible calls to NetSolve from Fortran. All these calls return a NetSolve code status. The list of the possible code status is given in Appendix D.

`CALL FNETSL('`*<problem name()>*`()',INFO, ...)`

Sends a **blocking** request to NetSolve. `FNETSL()` takes as argument the name of the problem, an integer, and the list of arguments in the calling sequence. See Section 4.3.2 for a discussion about this calling sequence. When the call returns, the integer `INFO` contains the NetSolve status code. If the call is successful, the result of the computation is stored in the output arguments. The output arguments are specified in the calling sequence.

`CALL FNETSLNB('`*<problem name()>*`()',INFO, ...)`

Sends a **nonblocking** request to NetSolve. `FNETSLNB()` takes as argument the name of the problem, an integer, and the list of arguments in the calling sequence. See Section 4.3.2 for a discussion about this calling sequence. It returns the NetSolve status code (integer `status`). If the call is successful, `status` contains the request handler.

`CALL FNETSLWT(`*<request handler>*`,INFO)`

Waits for a request completion. `FNETSLWT()` takes as argument a request handler and an integer. When the call returns, INFO contains the NetSolve status code. If the call is successful, the result of the computation is stored in the output arguments. The output arguments are specified in the calling sequence during the call to `FNETSLNB()`.

`CALL FNETSLPR(`*<request handler>*`,INFO)`

Probes for a request completion. `FNETSLPR()` takes as argument a request handler and an integer. When the call returns, INFO contains the NetSolve status code. If the call is successful, the result of the computation is stored in the output arguments. The output arguments are specified in the calling sequence during the call to `FNETSLNB()`.

`CALL FNETSLERR(`<*error code*>`)`

Displays an explicit error message given a NetSolve error code.

`CALL FNETSLMAJOR('`<*major*>`')`

Sets the way the user has stored her matrices (row- or column-wise). The argument can be `'col'` or `'row'`. It is case-insensitive and in fact only the first character is used by NetSolve.

Appendix D

Error Codes for C and Fortran

ERROR CODE	VALUE	MEANING
NetSolveOK	0	No NetSolve error
NetSolveNotReady	-1	Request still pending
NetSolveSetNetSolveAgent	-2	`NETSOLVE_AGENT` variable should be set
NetSolveSetNetSolveRoot	-3	`NETSOLVE_ROOT` variable should be set
NetSolveSetNetSolveArch	-4	`NETSOLVE_ARCH` variable should be set
NetSolveInternalError	-5	Internal NetSolve error
NetSolveUnknownHost	-6	Impossible to resolve hostname
NetSolveNetworkError	-7	Network error
NetSolveUnknownProblem	-8	Unknown problem nickname
NetSolveProtocolError	-9	Protocol error
NetSolveNoServer	-10	No server available for this computation
NetSolveBadProblemSpecification	-11	Bad input of output supplied
NetSolveNotAllowed	-12	Operation not allowed
NetSolveBadValues	-13	Bad input data supplied
NetSolveDimensionMismatch	-14	Matrix or Vector dimension mismatch
NetSolveNoSolution	-15	No solution for this problem
NetSolveUnknownError	-16	Unknown error
NetSolveInvalidRequestID	-17	Invalid request handle
NetSolveBadProblemName	-18	Invalid problem nickname
NetSolveInvalidMajor	19	Invalid major specification
NetSolveTooManyPendingRequests	-20	Too many pending requests
NetSolveFileError	-21	Error accessing/creating/deleting a file
NetSolveUnknownDataFormat	-22	Unknown host architecture
NetSolveInvalidUPFFilename	-23	Filename does not end in `.c` or `.f`
NetSolveMismatch	-24	Internal mismatch
NetSolveSystemError	-25	System error
NetSolveConnectionRefused	-26	Impossible to connect to host
NetSolveCannotBind	-27	Cannot bind to port
NetSolveUPFError	-28	Impossible to compile UPF
NetSolveUPFUnsafe	-29	UPF is unsafe
NetSolveServerError	-30	Server internal error
NetSolveBadIterationRange	-31	Bad farming range specification
NetSolveFarmingError	-32	One or more farming request failed

Appendix E

NetSolve Java API Reference

The Java API is being converted to NetSolve 1.2.

Appendix F

Complete C Example

```
#include "netsolve.h"
#define SIZE 100

main()
{
  double a[SIZE*SIZE];
  double x1[SIZE],y1[SIZE],x2[SIZE],y2[SIZE];
  int info,status;
  int i,init = 1325;

  for (i=0;i<SIZE*SIZE;i++) {
    init = 2315*init % 65536;
    a[i] = (double)((double)init - 32768.0) / 16384.0;
  }

   /* Setting the major */

   netslmajor("Col");

   /* NetSolve blocking */

   info = netsl("Eig()",a,SIZE,SIZE,x1,y1);
   if (info <0)
   {
     fprintf(stderr,"netsl() : %d\n",info);
     exit(0);
   }

  /* NetSolve Non-blocking */

  info = netslnb("Eig()",a,SIZE,SIZE,x2,y2);

  if (info < 0)
  {
    fprintf(stderr,"netslnb : %d\n",info);
    fprintf(stderr,"** NetSolve Abort **\n");
    exit(0);
  }
  status = netslwt(info);
```

```
  if (status <0)
  {
    fprintf(stderr,"netslwt() : %d\n",status);
    fprintf(stderr,"** NetSolve Abort **\n");
    exit(0);
  }
}
```

Appendix G

Complete Fortran Example

```
********************************************
*                                          *
* TEST of the FORTRAN INTERFACE to NETSOLVE*
*                                          *
********************************************

      PROGRAM TEST

      INCLUDE 'fnsolve.h'

      PARAMETER (SIZE = 2000)
      INTEGER N
      DOUBLE PRECISION A(SIZE,SIZE)
      DOUBLE PRECISION X1(SIZE)
      DOUBLE PRECISION Y1(SIZE)
      DOUBLE PRECISION X2(SIZE)
      DOUBLE PRECISION Y2(SIZE)
      INTEGER REQUEST
      INTEGER INFO
      INTEGER INIT,I,J

      INIT = 1325
      DO 10 I = 1,N
        DO 11 J = 1,N
          INIT = MOD(2315*INIT,65536)
          A(J,I) = (DBLE(INIT) - 32768.D0)/16384.D0
   11 CONTINUE
   10 CONTINUE

     CALL FNETSL('Eig()',INFO_NS,
     $                A,MAX,MAX,X1,Y1)

      IF(INFO_NS.LT.0) THEN
       CALL FNETSLERR(INFO_NS)
       STOP
      ENDIF

      CALL FNETSLNB('Eig()',REQUEST,
     $                A,MAX,MAX,X2,Y2)
```

```
IF(REQUEST.LT.0) THEN
 CALL FNETSLERR(REQUEST)
 STOP
ENDIF

CALL FNETSLWT(REQUEST,INFO_NS)
IF(INFO_NS.LT.0) THEN
 CALL FNETSLERR(INFO_NS)
 STOP
ENDIF

STOP
END
```

Appendix H

Complete Java Example

The Java API is being converted to NetSolve 1.2.

Bibliography

[1] The Math Works Inc. *MATLAB Reference Guide.* 1992.

[2] S. Wolfram. *The Mathematica Book, Third Edition.* Wolfram Median, Inc. and Cambridge University Press, 1996.

[3] H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[4] Henri Casanova and Jack Dongarra. NetSolve's Network Enabled Server: Examples and Applications. *IEEE Computational Science & Engineering*, 5(3):57–67, September 1998.

[5] H. Casanova and Dongarra. NetSolve version 1.2: Design and Implementation. Technical Report to appear, Department of Computer Science, University of Tennessee, 1998.

[6] H. Casanova, J. Dongarra, A. Karaivanov, and J. Wasniewski. Mathematica Interface to NetSolve. Technical Report UNIC-98-05, UNI-C, September 1998.

[7] M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 104–111. Department of Computer Science, University of Winsconsin, Madison, June 1988.

[8] M. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. In *Proc. of IEEE Workshop on Experimental Distributed Systems.* Department of Computer Science, University of Winsconsin, Madison, 1990.

[9] H. Casanova and J. Dongarra. Providing Uniform Dynamic Access to Numerical Software. In M. Heath, A. Ranade, and R. Schrieber, editors, *IMA Volumes in Mathematics and its Applications, Algorithms for Parallel Processing*, volume 105, pages 345–355. Springer-Verlag. to appear in 1998.

[10] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995.