Users' Guide to NetSolve

version 1.1.b
(Client and Server)

Henri Casanova[1] Jack Dongarra[1 2] Keith Seymour[1]

March 31, 1998

[1]Department of Computer Science, University of Tennessee, TN 37996
[2]Mathematical Science Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831

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 four chapters. The first chapter presents the basic concepts of NetSolve and gives references to several publications concerning the project. The second chapter describes the client side of NetSolve, reviewing the Matlab, C, Fortran and Java Application Programming Interfaces (APIs) as well as the Java Graphical User Interface (GUI). That section also gives several illustrative examples. The third chapter describes the server side of NetSolve and gives details on each of its software components. The fourth and last chapter describes how NetSolve handles the *user-provided function* mechanism. Finally, complete reference manuals 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. MATLAB [1] is an example of such a tool.
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, or the World Wide Web can easily use the NetSolve system thanks to the different client types.

1.2 Overview of the NetSolve System

1.2.1 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

Solving a problem with NetSolve is done in three steps. The client sends a request to the agent. The agent chooses the "best" NetSolve resource according to the size and nature of the problem to be solved. The problem is then solved on the chosen server, and the result is sent back to the client.

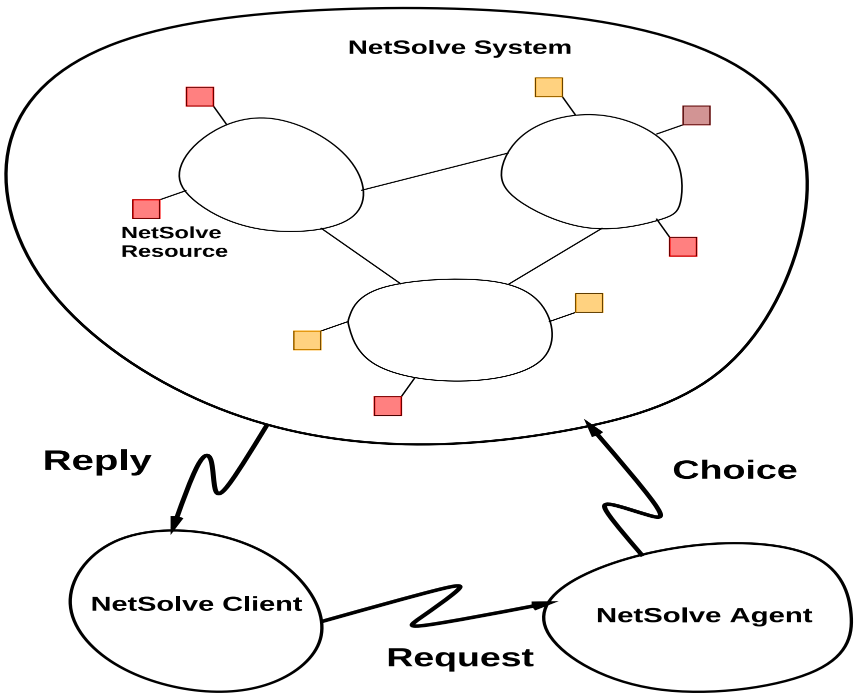


Figure 1.1: The NetSolve System

This system is fault tolerant, meaning that the client will receive an answer to its problem 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 [2].

1.2.2 Problem Specification

To keep NetSolve as general as possible, we needed a formal way of describing a problem.
A problem is defined as a 3-tuple: $< name, inputs, outputs >$, where

- *name* is a character string containing the name of the problem,
- *inputs* is a list of input objects, and
- *outputs* is a list of output objects.

An object is itself described as follows: $< object, data >$, where *object* can be MATRIX, VECTOR, or SCALAR, and *data* can be any of the standard Fortran data types.
This description has proved to be sufficient to interface NetSolve with numerous software packages. NetSolve is still at an early stage of development and is likely to undergo modifications in the future. For the time being, the existing interfaces use this formalism. However, we will see that the computational servers are usually set up so that the calling sequences to be used by the NetSolve APIs fit the underlying scientific software calling sequences.

1.2.3 Problems that can be solved with NetSolve

Before actually using NetSolve with any interface, the user needs to know what problems are solvable. The easiest way is to check the NetSolve homepage:

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

The *Available Resources* page provides access to two CGI scripts. Using those scripts, one can inquire about which problems are handled by the servers and about which servers are in the system. Those scripts give complete details for the C and Fortran interfaces. This information is also available from the Java or the MATLAB interfaces, for which such a level of detail is not required. In the future, we plan to suppress those scripts and replace them with a Java applet. This Java applet will look very similar to the current NetSolve Java interface and will provide information only about the C and Fortran interfaces.

This early version of NetSolve has a naming scheme for problems. We can distinguish the *name* of a problem and its *full name*. The full name has a path-like structure. Let us explain this with an example. The problem `ddot`, which computes the inner product of two double-precision vectors, can have a full name like `/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 comes from. This convention has proven to be useful, as seen in Section 2.3.2.

Chapter 2

The NetSolve Client

2.1 Getting Started

2.1.1 Downloading the Software

The client software can be downloaded from the NetSolve homepage at

`http://www.cs.utk.edu/netsolve/client_distribution.tar.gz`.

The following UNIX commands will create the `Netsolve_client` directory:

```
% gunzip client_distribution.tar.gz
% tar -xvf client_distribution.tar
```

2.1.2 Setting the Architecture

The `Netsolve_client` directory includes a Shell script called `netsolvegetarch` that can be used to return a character string describing the architecture of the user's machine. 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
```

or preferably, if `netsolvegetarch` is in the path:

```
setenv NETSOLVE_ARCH `netsolvegetarch`
```

To date, NetSolve has been ported to the following different architectures:

- 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 IRIS running OS $\geq$ 5.0
- SGI64 : Silicon Graphics IRIS running OS $\geq$ 6.0
- HPPA: HP 9000 PA-Risc
- RS6K: IBM RS/6000

2.1.3 Compiling

Now that the `NETSOLVE_ARCH` environment variable has been set as described in Section 2.1.2, the software can be compiled. First, one should go to the `Netsolve_client/conf` directory and edit the `$NETSOLVE_ARCH.def` file (for instance `RS6K.def`). This file contains a custom section in which the user can modify the compilation parameters. Here is a typical section:

```
# ---- Custom Section ----
F77             = f77
CC              = cc
CMEX            = cmex
MATLAB_VERSION  = 5
# ---- End of Custom Section ----
```

This custom section specifies which compilers are going to be used. `CMEX` denotes the MATLAB C-compiler, in case the MATLAB interface is to be built. `MATLAB_VERSION` can take the value `4` or `5` depending on which version of Matlab is to be used. These parameters can be modified before compilation. However, the file also contains other information that should not be modified. The NetSolve clients can now be compiled. Typing `make` in the `Netsolve_client` directory will give instructions to complete the compilation.

2.1.4 Setting an Agent Name

As described in Section 1.2.1, to solve a problem, a client must contact an agent. The C, Fortran, and MATLAB 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 some 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 `comet.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 comet.cs.utk.edu
```

The Java GUI and API do not require the `NETSOLVE_AGENT` environment variable.

2.1.5 Testing

The thing to do at this point is to test the client software by typing 'Test' in the NetSolve client directory. This test generally takes a few minutes but may take longer depending on your distance to the agent. We advise to test the client with the servers running at the University of Tennessee by setting the `NETSOLVE_AGENT` environment variable to `comet.cs.utk.edu`.

2.2 MATLAB Interface

2.2.1 Introduction

Building the MATLAB interface as described in Section 2.1.3 produces the two following *mex-files* :

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

The `###` part of the extension depends on the architecture (for instance, the extension is `.mexsol` for the Solaris Operating System). These two files alone are the MATLAB interface to NetSolve. Modifying the `MATLABPATH` environment variable will make these two files available from any location in MATLAB. For more information about mex-files, the user can refer to [3]. Basically, the user will now be able to call two new functions from MATLAB: `netsolve()` and `netsolve_nb()`. The following sections will explain how to use those two functions.

2.2.2 What to Do First

Let us now assume that the user has started a MATLAB session and is ready to try NetSolve. In this section we describe those features of this interface that allow the user to get information about the currently available NetSolve system.
As stated briefly in Section 1.2.3, 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. This list can be really long, and in that case it is wiser to use the CGI scripts in Section 1.2.3. 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)
```

This is the same kind of information as that available from the CGI scripts. It gives a short description of what the problem is. Usually it also includes an example for MATLAB, using `netsolve()`.

```
---------
- INPUT -
---------
 #0 : Double-precision real matrix.
 Matrix A
```

This is the description of the input the user needs to give NetSolve. This particular problem requires only one double-precision matrix. 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.

```
----------
- OUTPUT -
----------
 #0 : Double-precision real vector.
 Real parts of the eigenvalues
 #1 : Double-precision real vector.
 Imaginary parts of the eigenvalues
```

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

```
----------------------------------------------
Output 0 and 1 can be merged to form a complex object
----------------------------------------------
```

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 following 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 `comet.cs.utk.edu` (see Section 2.1.4). Let us try the following command:

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

this command produces the following output :

```
comet.cs.utk.edu (128.169.92.78)
        NetSolve Agent
        Host : Up       Server : Running
maruti.CS.Berkeley.EDU (128.32.36.83)
        Handles 10 problems
        Host : Up       Server : Running
cupid.cs.utk.edu (128.169.94.221)
        Handles 29 problems
        Host : Up       Server : Running
```

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

1. `comet` itself, which is stated as being an *agent*
2. `cupid` at the same location, which is a computational server handling 29 different problems
3. `maruti` at U.C. Berkeley, which is also a computational server and handles 10 different problems

We can also see the status information about the servers (the processes) and the hosts (the computers). Right now, everything is up and running.

2.2.3 Calling `netsolve()`

The first 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; this approach is described in Section 2.2.4.
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 2.2.2). 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 2.2.2). 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 this command is typed:

```
>> [x y] = netsolve('eig',a)
Trying server cupid.cs.utk.edu
Problem accepted....sending the data
Waiting for result.....
Result received

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)
Trying server cupid.cs.utk.edu
Problem accepted....sending the data
Result received

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

2.2.4 Calling `netsolve_nb()`

The obvious drawback of the function `netsolve()` is that while the computation is performed remotely, the user must simply wait to get back the prompt. To address this drawback, we designed `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 are going to be solved on different machines, achieving some *NetSolve-parallelism.* Let us now describe this function on the `eig` example.
As in Section 2.2.3, 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 quite 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)
Trying server cupid.cs.utk.edu
Problem accepted....sending the data

r =
    0
```

As expected, `netsolve_nb()` returns a request handler: here it is `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 :

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

The left-hand side of this call is the left-hand side of the call to `netsolve()`. The right-hand side contains the action, as is required for `netsolve_nb()`, and the request handler. This call returns immediately, either printing out a message saying that the result has not arrived yet or giving the result in `x` and `y`. Here are the two possible scenarios:

```
>> [x y] = netsolve_nb('probe',r)
Not ready yet
>> ... Some other work ...
>> [x y] = netsolve_nb('probe',r)
Result received

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

The other way to obtain the result is to call `netsolve_nb()` with the `'wait'` action. The call then blocks until the result arrives:

```
>> [x y] = netsolve_nb('wait',r)
Waiting for result.....
Result received

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

As for `netsolve()`, we 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'`.
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)
Trying server cupid.cs.utk.edu
Problem accepted....sending the data
r1 =
     0
>> [r2] = netsolve_nb('send','eig',b)
Trying server vw.cs.Berkeley.edu
Problem accepted....sending the data
r2 =
     1
```

Now let us see what `status` does:

```
>> netsolve_nb('status')
Pending NetSolve requests :
Request #0 - eig
        Assigned to cupid.cs.utk.edu 12 seconds ago
        Still RUNNING
        Predicted execution time  : 304 seconds
```

```
Request #1 - eig
        Assigned to vw.cs.Berkeley.edu 3 seconds ago
        Still RUNNING
        Predicted execution time  : 402 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.

2.2.5 What Can Go Wrong?

During a computation, two classes of error can occur: NetSolve failures and user mistakes.

NetSolve Failures

The first class of error is caused by the NetSolve system itself, that is, the pool of agents and servers. The `netsolve()` and `netsolve_nb()` functions print out explicit and simple error messages, and we are not going to describe them all in great detail. Let us mention just one:

```
>> netsolve
No agent running on demidoff.cs.utk.edu
```

The environment variable `NETSOLVE_AGENT` contains the name of a host that is not running a NetSolve agent. All the other messages are of the same form and easily understandable.

User Mistakes

The second class of error comes from the user. If the user does not follow the calling sequences described in Sections 2.2.3 and 2.2.4, error messages are printed out. For instance, if the user passes a problem name that does not exist, NetSolve will indicate that this problem is unknown at this time. Again, all the messages are explicit, and we are not going to list them all here.
More interesting errors occur when the calling sequences are respected but the user provides *wrong* data to NetSolve. Here is an example of such a case:

```
>> a = rand(300,400)
>> [x] = netsolve('eig',a)
Trying server cupid.cs.utk.edu
Problem accepted....sending the data
** Dimension mismatch **
x =
    []
```

The user tried to compute the eigenvalues of a nonsquare matrix, and NetSolve indicates that the computation is impossible. The same kind of message is printed for any mistake in the input data.

2.3 C and Fortran Interfaces

2.3.1 Introduction

The C and Fortran interfaces are, in fact, one. The Fortran interface is built on top of the C interface, since all the networking underneath NetSolve is done in C. However, we chose to design the Fortran wrappers around the C interface as subroutines (instead of functions). The C functions all return an integer called the *NetSolve status code*. The Fortran subroutine just takes it as an argument passed by reference. The list

of all the possible NetSolve status codes can be found in Appendix D. The reference manuals for C and Fortran are in Appendixes B and C.
The basic concepts here are the same as the ones we have introduced in Section 2.2 for the MATLAB interface, especially the ability to call NetSolve in a blocking or nonblocking fashion.
After compiling the C/Fortran interface as explained in Section 2.1.3, the user will find two archive files:

- `Netsolve_client/lib/$NETSOLVE_ARCH/libnetsolve.a` : the C library
- `Netsolve_client/lib/$NETSOLVE_ARCH/libfnetsolve.a` : the Fortran library

The user must link his C or Fortran program to either one of these libraries to enable it to call NetSolve. The user must also include the following header file:

- `Netsolve_client/include/netsolve.h` in C,
- `Netsolve_client/include/fnetsolve.h` in Fortran.

Before describing the interface itself, we discuss the calling sequence to use for the different problems in the next section.

2.3.2 Knowing the Calling Sequence

When we described the MATLAB interface in Section 2.2, the calling sequence of `netsolve()` was fairly simple. It consisted of the input objects on the right-hand side and the output object on the left-hand side. On each side, the objects were in the same order as the one they were listed in the problem description. Since this problem description is available from MATLAB, the user could easily determine the proper calling sequence. The situation is not that simple for C or Fortran. Indeed, MATLAB is a high-level computational tool that provides its users with high-level objects encapsulating several pieces of data. For instance, in MATLAB a matrix is an object that can be referenced with a single identifier, even though it contains two integers, and a pointer to an array of double-precision elements. The two integers, of course, are the number of rows and columns of the matrix, and the pointer points to the element of the matrix (stored columnwise in MATLAB). Hence, when a user passes a matrix identifier to NetSolve from MATLAB, he does not have to worry about passing the sizes of the matrix.
In C or Fortran, we do not have access to such high-level constructs. Therefore, when we pass to NetSolve a pointer to some data, we also need to specify the size(s) of this data. This requirement, of course, implies that the calling sequence has to be more complex than the one in MATLAB. In Section 1.2.3, we noted that the CGI scripts were giving extensive details about the different problems. Those details are, in fact, the descriptions of the C and Fortran calling sequences.
Our present policy with calling sequences from C or Fortran is to preserve the native calling sequences of the numerical software. Recall that in Section 1.2.3, we said that, by convention, the first element of the full name of a problem is the name of the numerical library the problem comes from. Therefore, the user always knows what software a routine comes from, by consulting the NetSolve homepage.
Thus, two situations are possible. First, the user knows the numerical software and may even have a code already written in terms of this software. Then, *switching* to NetSolve is immediate, and we will see examples in the following sections. The second possibility is that the user does not know the software. Then he can learn the calling sequences from the NetSolve homepage thanks to the CGI scripts. The NetSolve homepage will also give access to URLs that may contain information about the different software in use.
With this understanding of how calling sequences work, we can proceed with the actual description of the interface.

2.3.3 Blocking Call

As with MATLAB, there is a blocking call to NetSolve from C or Fortran. Specifically, one calls a single function, `netsl()`. This function returns a NetSolve status 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 2.3.2 and their number of variables. The C prototype of the function is

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

and the Fortran prototype is

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

where `PROBLEM_NAME` is a string and `NSINFO` is the status code returned by NetSolve. The number of the arguments in the calling sequence depends on the problem.
Let us consider an example that uses the LAPACK [4] routine `dgesv()`, which solves a linear system of equations. In Fortran, the direct call to LAPACK looks like

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

The equivalent blocking call to NetSolve is

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

The call in C is

```
nsinfo = 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, this workspace is useless on the client side. 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!

2.3.4 Nonblocking Call

We developed this nonblocking call for the same reason we developed one for MATLAB (see Section 2.2.4): to allow the user to have some *NetSolve-parallelism*. The nonblocking version of `netsl()` is called `netslnb()`. The user calls it in **exactly** the same way `netsl()` is called. The only difference between the two functions lies in the NetSolve status code they return. If the call to `netslnb()` is successful, a request handler is returned in the NetSolve status code, as in the MATLAB interface. Let us give an example in Fortran:

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

and in C :

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

This is exactly the same call as the one in the preceding section.
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()`. If the call is successful, the function returns immediately with either a NetSolve status code telling that the result is not available yet or with the result in the user space. Here is an example in Fortran:

```
      call FNETSLPR(REQUEST,NSINFO)
```

and in C :

```
nsinfo = netslpr(request);
```

Waiting is done by using `netslwt()`. This function blocks until the request is completed. Here is the Fortran call:

```
      call FNETSLWT(REQUEST,NSINFO)
```

and the C call :

```
nsinfo = netslwt(request);
```

If the call is successful, the function returns with the results in the user space.

2.3.5 Error messages

There is an additional 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(nsinfo);
```

and in Fortran

```
       call FNETSLERR(NSINFO)
```

2.3.6 Row- or Column-major

To allow the NetSolve user to store her 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.

2.3.7 Built-in Examples

C and Fortran and Java examples are included in the NetSolve Client Distribution in the directory `Netsolve_client/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 a full example in C and Fortran in Appendixes F and G.

2.4 Java API

2.4.1 Introduction

The Java API to NetSolve is designed to give Java application programmers the ability to access NetSolve resources from their programs. This allows access to a wide variety of numerical software that has not yet been implemented in Java. Unfortunately we were not able to create a NetSolve API that was identical to the C and Fortran interfaces because they rely on the ability to write functions that accept a variable number of parameters. The Java language does not provide this ability, so we had to devise another interface. There are a couple of ways to provide variable-length argument lists in Java. First, the user could pack all the input items in an array of `Objects` and pass the array as the only argument to the NetSolve API. Alternately, the API could provide a function that lets the programmer specify one argument at a time. We chose to implement the second method since it requires the least effort from the user and it allows the API to perform better error checking of argument-parameter mismatch.
Other than the method in which arguments are passed, the basic functionality of the NetSolve API matches that of the C and Fortran interfaces, including blocking and nonblocking calls to NetSolve. See Appendix E for the NetSolve API reference manual. Also, there is a full example of using the NetSolve API in Appendix H.
After compiling the source code that comprises the NetSolve API, the user should set the `CLASSPATH` environment variable to include the directory in which the API class files reside. Typically, the `CLASSPATH` is set as follows:

```
setenv CLASSPATH .:/home/user/Netsolve_client/src/java
```

The `.cshrc` file is a good place to set the `CLASSPATH`. For shells other than csh, the procedure may be different.
Once the `CLASSPATH` has been set, the user can write and compile source code containing calls to the NetSolve API. Keep in mind that the Java source code comprising the API is written using features of version 1.1 of the Java Development Kit (JDK). Therefore, it will be necessary to have version 1.1 or newer installed in order to use the NetSolve API.

2.4.2 Establishing a Connection

The first thing that must be done in order to access NetSolve resources from a Java program is to establish a connection to the agent using the `NetsolveSession` class. The user provides the name of the machine on which the agent is running. If no name is provided, a default agent is contacted. For example,

```
ns = new NetsolveSession("woodstock.cs.utk.edu");
```

would contact the agent running on `woodstock.cs.utk.edu`, while

```
ns = new NetsolveSession();
```

would contact the default agent (currently set to `comet.cs.utk.edu`).
Contacting the agent serves two purposes. First, it ensures that the agent is currently available. Second, it allows the API to maintain a list of all the problems that the agent can solve. Since retrieving the problem list can be time-consuming, the same `NetsolveSession` can be reused without having to reload the problem list as long as the same agent is going to be contacted. However, if the user program must switch agents, a new `NetsolveSession` must be created. `NetsolveSession` does not maintain a persistent connection to the agent. It is merely an encapsulation of the agent's hostname and the list of problems.

2.4.3 Knowing the Calling Sequence

After a `NetsolveSession` has been established, the problem name and parameters are specified. As with the C and Fortran interfaces, users of the NetSolve API must know the calling sequence of the problem they

wish to solve. Information about the number and type of input parameters can be obtained through the main screen of the Java GUI, discussed in Section 2.5.
The first step is to create a new `Netsolve` object, specifying a previously created `NetsolveSession`:

```
ns = new NetsolveSession("woodstock.cs.utk.edu");
n = new Netsolve(ns);
```

Passing the session to `Netsolve` lets the API know to which agent the data should be sent and which problems the agent can solve.
Then, the problem name and arguments are specified:

```
n.setProblem("dgesv");
n.pushArg(matrixA);
n.pushArg(matrixB);
```

The problem name, `dgesv`, is specified first and then the arguments are pushed one at a time, in order. Since Java allows the `pushArg()` method to be overloaded, the same method can be called regardless of data type.
Once the problem name and parameters have been specified and no errors have been detected (see Section 2.4.7), the problem may be submitted. The NetSolve API provides for blocking and nonblocking calls, described in the following two sections (2.4.4 and 2.4.5, respectively).

2.4.4 Blocking Call

To have the data sent to the server and begin the computation, the `submitProb()` method of a `Netsolve` object is called. This begins the computation in *blocking* mode. That is, the call to `submitProb()` does not return until the entire computation has finished and the results have been obtained. To continue with the previous example, let us assume that the user has specified the problem (`dgesv`) and passed both input matrices to the `Netsolve` object. Now, to begin the computation, all that remains is to submit the data, as follows:

```
n.submitProb();
```

No parameters are needed since all data has already been stored in the `Netsolve` object. Similarly, after the call to `submitProb()` returns, the results of the computation are also stored in the `Netsolve` object. Section 2.4.6 discusses how to retrieve the results.

2.4.5 Nonblocking Call

As with the other NetSolve interfaces, the NetSolve API also provides a *nonblocking* call. The problem specification is exactly the same as with the blocking version. The only difference is that instead of calling `submitProb()`, the user calls `submitProbNB()`, as follows:

```
n.submitProbNB();
```

The call to `submitProbNB()` returns immediately, allowing the user to perform other computations while the problem is being submitted and solved. Once these other computations have completed, the user may wait for the results from a particular computation by calling the `waitFor()` method of the `Netsolve` object that submitted it. For example:

```
n.submitProbNB();
n2.submitProbNB();

// do other computations

n2.waitFor();   // wait for the second computation
n.waitFor();    // wait for the first computation
```

As the preceding example illustrates, it is easy to distinguish between the two submissions since they are encapsulated in two different objects.

2.4.6 Retrieving the Results

After the results of the computation are received from the NetSolve server, they are stored into a `Vector` object within the `Netsolve` object that submitted the job. To retrieve this data, the user calls the `getOutput()` method, as follows:

```
Vector out;

ns = new NetsolveSession("comet.cs.utk.edu");

n = new Netsolve(ns);      // Specify 'session'
n.setProblem("dgesv");     // Specify the problem to solve
n.pushArg(a1);             // Pass first parameter to dgesv
n.pushArg(a2);             // Pass second parameter to dgesv
n.submitProb();            // Submit this problem
out = n.getOutput();       // Get the output item(s)
```

Notice on the last line that `getOutput()` returns a Vector, which we assign to a local variable `out`. Each element of the Vector contains one output item. In this case, the problem (`dgesv`) returns four output items:

- Double-precision matrix
- Integer Vector
- Double-precision matrix
- Integer Scalar

Therefore, we first create four local variables to hold the results:

```
Integer outInt;
double [][] outMat1, outMat2;
int [] outVec;
```

Note that, as with all scalars, the integer scalar will be returned wrapped in an object. So, we declare `outInt` to be of type `Integer`. Finally, to assign the output items to the local variables requires acessing the individual element of the Vector, using the `elementAt()` method.

```
outMat1 = (double [][]) (out.elementAt(0));
outVec  =      (int []) (out.elementAt(1));
outMat2 = (double [][]) (out.elementAt(2));
outInt  =     (Integer) (out.elementAt(3));
```

Note that the return type of `elementAt()` is `Object`, which we must cast to the appropriate type in the assignment statement. The preceding example demonstrates that the user must have detailed knowledge of the output items returned by the problem. This information can be obtained by consulting the initial screen of the Java GUI (as described in Section 2.5).

2.4.7 Error messages

Errors may occur at many stages of the job submission process. each method in the NetSolve API throws a `NetSolveException` upon any error condition. As the following example shows, this can simplify error handling in the user's program by moving all error handling code to one location:

```
try {
  ns = new NetsolveSession("comet.cs.utk.edu");

  n = new Netsolve(ns);     // Specify 'session'
  n.setProblem("dgesv");  // Specify the problem to solve
  n.pushArg(a1);            // Pass first parameter to dgesv
  n.pushArg(a2);            // Pass second parameter to dgesv
  n.submitProb();           // Submit this problem
  out = n.getOutput();      // Get the output item(s)

}catch(NetSolveException e) {
  System.err.println("Error in submission:");
  System.err.println(e.getMessage());
  System.exit(1);
}
```

When the API detects an error in problem specification or job submission, it creates a description of the nature of the error. This error string is accessible through the `getMessage()` method of the `NetSolveException` object.
Of course, the user still has the option to perform error checking after each stage of the submission process in case more specific error handling actions are necessary.

2.4.8 Row- or Column-major

While most Java programmers will store their matrices in row-major format, the NetSolve API provides the ability to switch between row-major and column-major representations. Naturally, one of the following functions should be called prior to submitting the data:

```
n.setMajor("row");
n.setMajor("col");
```

By default, row-major is assumed, so the user does not need to call `setMajor()` unless switching to column-major.

2.4.9 Built-in Examples

Several examples of using the NetSolve API can be found in `Netsolve_client/examples/JavaAPI`. As with the C and Fortran examples, the examples use problems that can be solved on servers at the University of Tennessee. There is also a full example Java program in Appendix H.

2.5 Java GUI

2.5.1 Introduction

This section describes the Java interface to NetSolve, a user-friendly graphical tool for accessing resources in the NetSolve system. Since the Java interface should be runnable from many WWW browsers, it also provides users the opportunity to solve problems without downloading or compiling any source code. However, the current Web browser versions impose very strong restrictions on the capabilities of applets. At this time, it appears to be impossible to open sockets to a remote host that is not running the Web server. Additionally, the latest version of the Java interface (GUI and API) were developed using version 1.1 of the Java Development Kit (JDK). Because not all browsers support the new features in version 1.1, the NetSolve Java interface will not be compatible with every browser. Future versions of these web browsers

will undoubtedly alleviate these problems. For the time being, a demo applet is available on the NetSolve homepage. It uses an agent and a server that are both running on the Web server.
To start the stand-alone application:

```
java NetSolveClient comet.cs.utk.edu
```

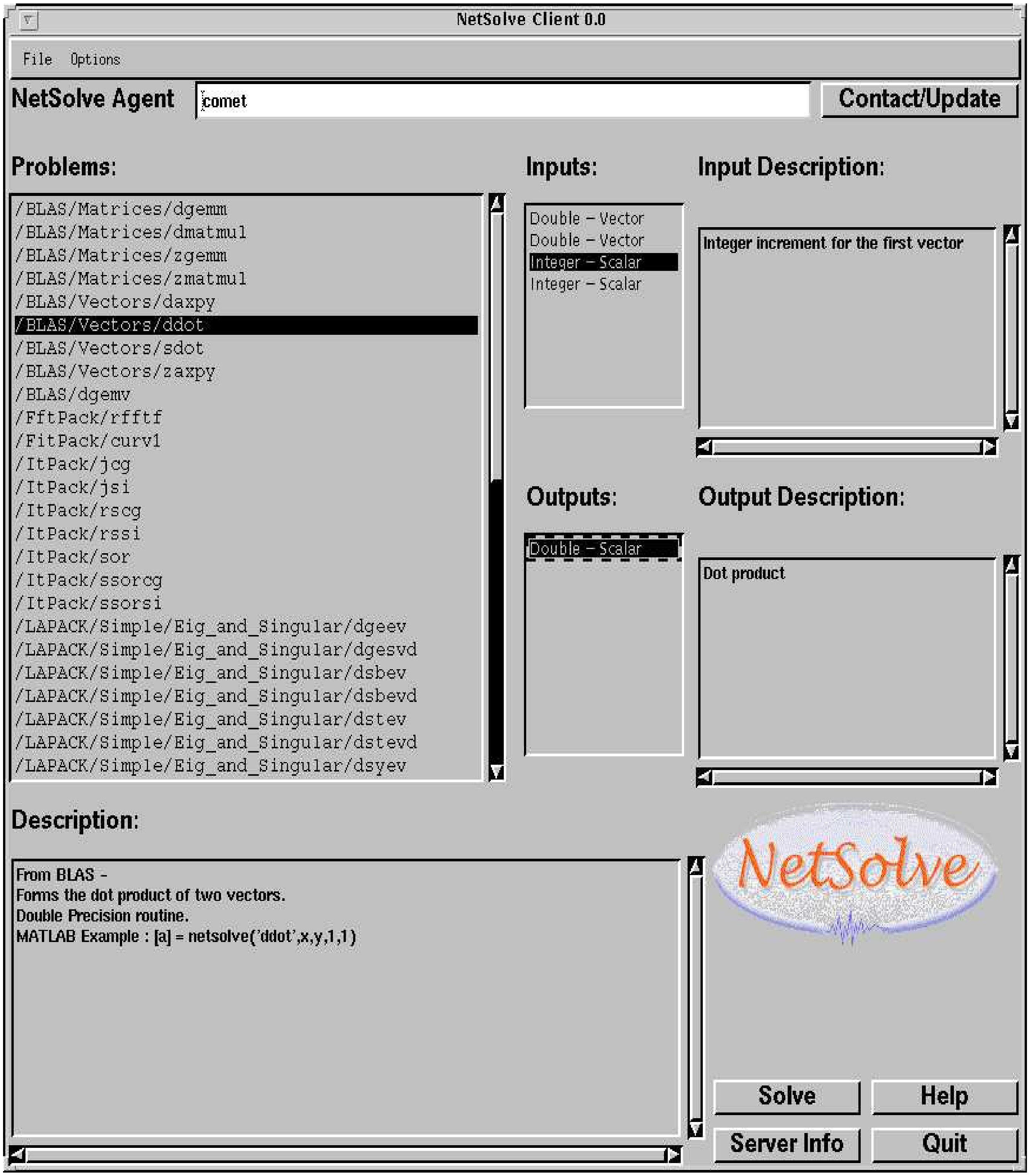
where comet.cs.utk.edu is the name of a machine running a NetSolve agent. The machine name is optional, but if it is not specified, the client tries to contact `comet.cs.utk.edu` by default.

2.5.2 The Initial Screen

Let us now assume that the user has started the Java interface, either as an applet (via the Web) or as a stand-alone application. Figure 2.1 shows the initial screen, which consists of several components:

- Agent Selection Box
- Problem List
- Problem Description Box
- Input List
- Input Description Box
- Output List
- Output Description Box

To contact an agent, the user can enter the hostname in the *Agent Selection Box* and then click on the "Contact/Update" button. In some cases, the user may have already contacted an agent, but just wants to update the list of problems. If so, clicking on the "Contact/Update" button without changing the text in the *Agent Selection Box* will reload the problem list. Once the list of available problems has been loaded it is then displayed in the *Problem List*, located in the upper left region of the interface.



comet

/BLAS/Matrices/dgemm
/BLAS/Matrices/dmatmul
/BLAS/Matrices/zgemm
/BLAS/Matrices/zmatmul
/BLAS/Vectors/daxpy
/BLAS/Vectors/ddot
/BLAS/Vectors/sdot
/BLAS/Vectors/zaxpy
/BLAS/dgemv
/FftPack/rfftf
/FitPack/curv1
/ItPack/jcg
/ItPack/jsi
/ItPack/rscg
/ItPack/rssi
/ItPack/sor
/ItPack/ssorcg
/ItPack/ssorsi
/LAPACK/Simple/Eig_and_Singular/dgeev
/LAPACK/Simple/Eig_and_Singular/dgesvd
/LAPACK/Simple/Eig_and_Singular/dsbev
/LAPACK/Simple/Eig_and_Singular/dsbevd
/LAPACK/Simple/Eig_and_Singular/dstev
/LAPACK/Simple/Eig_and_Singular/dstevd
/LAPACK/Simple/Eig_and_Singular/dsyev

Double – Vector
Double – Vector
Integer – Scalar
Integer – Scalar

Integer increment for the first vector

Double – Scalar

Dot product

From BLAS -
Forms the dot product of two vectors.
Double Precision routine.
MATLAB Example : [a] = netsolve('ddot',x,y,1,1)

Figure 2.1: The Initial Screen

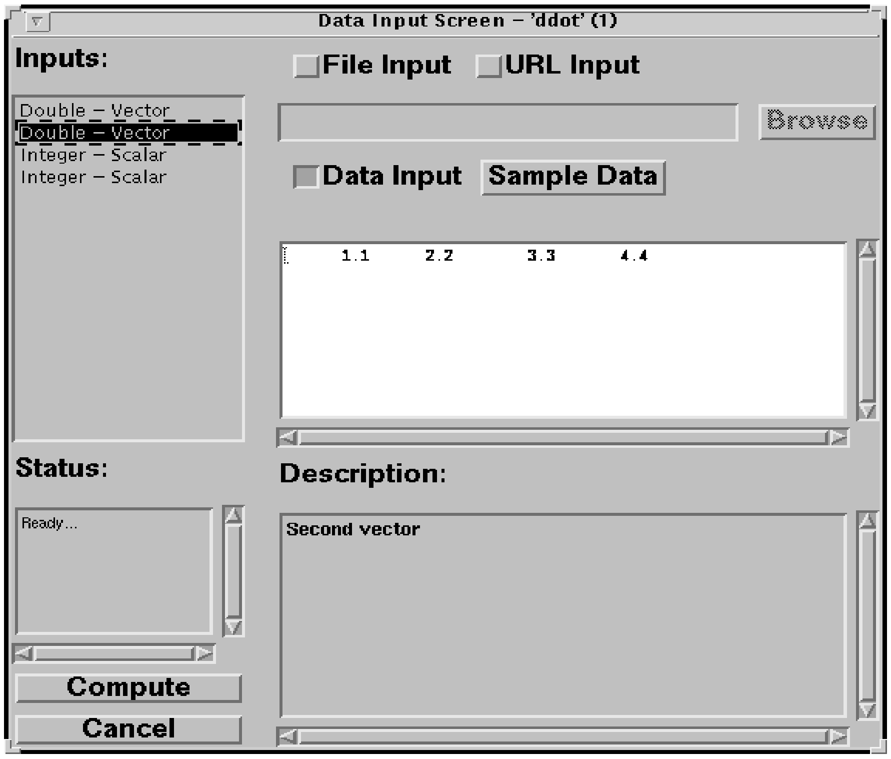
To find out more about any problem listed, the user may click on that problem and view pertinent information displayed in the *Problem Description Box*, the *Input List*, and the *Output List*. The *Problem Description Box*, located in the lower left region of the interface, contains a short description of the selected problem. The *Input List* contains a list of the input objects required to solve the selected problem. Similarly, the *Output List* contains a list of the output objects that are returned by the server. When the user clicks on any item in the *Input List*, the interface updates the *Input Description Box* with text describing the selected input object. Likewise, clicking on any item in the *Output List* updates the *Output Description Box* with text describing the selected output object.

2.5.3 Solving a Problem

To solve an instance of some problem, the user must first select a problem from the *Problem List* and then click on the "Solve" button (alternately, the user may double-click on an item in the *Problem List*). A new window will appear allowing the user to input data for each input object required by the problem. Figure 2.2 shows the *Data Input Window*, which consists of the following components:

- Input List
- Input Description Box
- Filename (or URL) Selection Box
- Data Input Box
- Status Box

The *Input List* contains a list of the input objects for which the user must supply data. The *Input Description Box* contains text describing the selected input object (this text is the same as the text displayed in the *Input Description Box* of the initial screen).
For each input object, the user may choose to enter the data manually into the *Data Input Box* or specify the name of a file or URL containing the data in the *Filename/URL Selection Box*. Since the same input box is used for both filenames and URLs, the user must specify whether the string in the text box should be treated as a filename or a URL by clicking on one of the two checkboxes above the *Filename/URL Selection Box*. Next to the *Filename/URL Selection Box* is a "Browse" button which allows choosing the file using a graphical file browser. The file browser is only available when the user has selected to load the data from a file. Just above the *Data Input Box* is a "Sample Data" button which fills the box with some numbers appropriate to the type of the input object (for example, if the input object is a vector of integers, clicking on the "Sample Data" button will generate a vector of integers). Note that even though the interface allows having text in both selection boxes simultaneously, only one box may be "active" at any time and anything in the "inactive" box will be ignored. The user may easily distinguish between the two boxes since the inactive box has a grey background and the active box has a white background. In addition, the checkbox adjacent to the active box will appear depressed.
The title bar of the *Data Input Window* contains some noteworthy information: the name of the problem, and a *Request Number*. The problem name listed on the title bar is the same name from the initial screen, minus the path. For example, if the full name as shown on the initial screen is `/Blah/blah/prob`, then the name on the title bar is `prob`. The *Request Number* is a number which uniquely identifies each *Data Input Window* so that the user may easily relate the *Output Windows* (see Section 2.5.4) to the *Input Windows* from which they originated.
Once all inputs have been fully specified, click on the "Compute" button, located in the lower left region of the *Data Input Window*. If there are any errors in the data and/or files, an informational window will appear describing the nature of the errors and for which input object(s) the errors apply. All errors must be corrected before the data may be sent. Here are some of the most common errors:



Double – Vector
Double – Vector
Integer – Scalar
Integer – Scalar

```
1.1    2.2    3.3    4.4
```

Ready...

Second vector

Figure 2.2: The Input Screen

- Invalid numeric format. The input does not match the expected input type (for example, the input type is "integer" and the user enters "1.2").
- Empty input. The user did not specify any data for some input object.
- Input not specified. This is similar to the previous error except that here, the user did not activate one of the two input sources (file input or data input) whereas in the previous error, one of the two input boxes was chosen, but no data was entered.
- Nonexistent file. The filename given does not exist. Using the graphical file browser may help determine the correct path and file name.
- Rows of matrix not even. This means that one or more rows in the matrix do not have the same number of elements.

If the data and/or files specified are acceptable, the values are sent to a computational server which performs the computations and returns the output objects.
The *Status Box* provides information about the current status of the data submission. As the job progresses, it is updated with brief messages stating, for example, that the agent is being contacted, that the data is being sent, and so on.

2.5.4 Viewing the Results

Once the computational server sends back the results, a new window appears allowing the user to browse the results. Figure 2.3 shows the *Output Window*, which consists of the following components:

- Output List
- Output Description Box
- Data Box

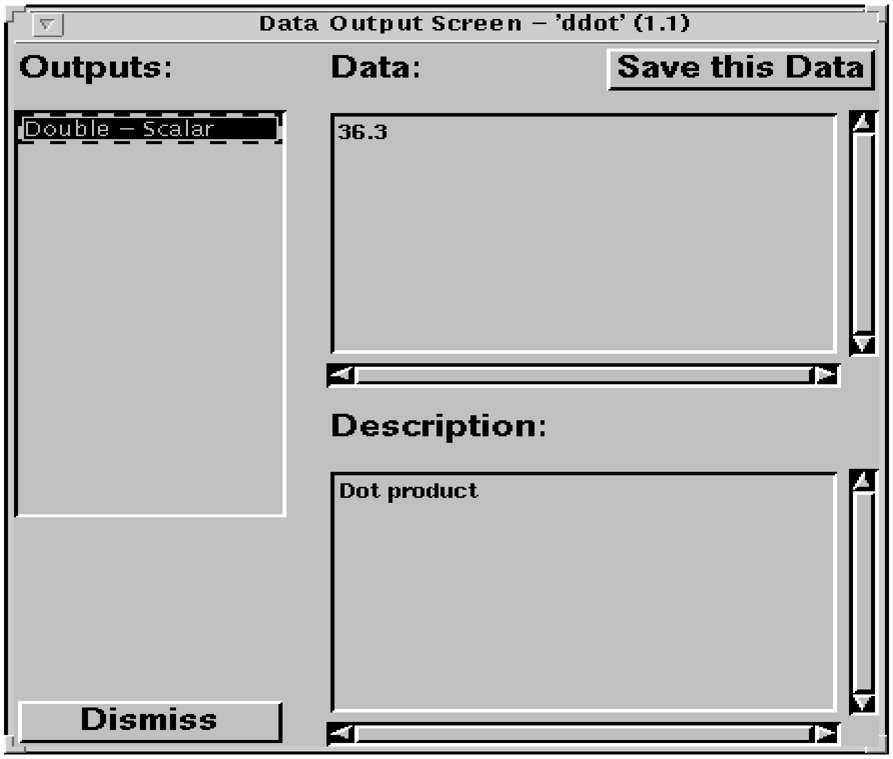
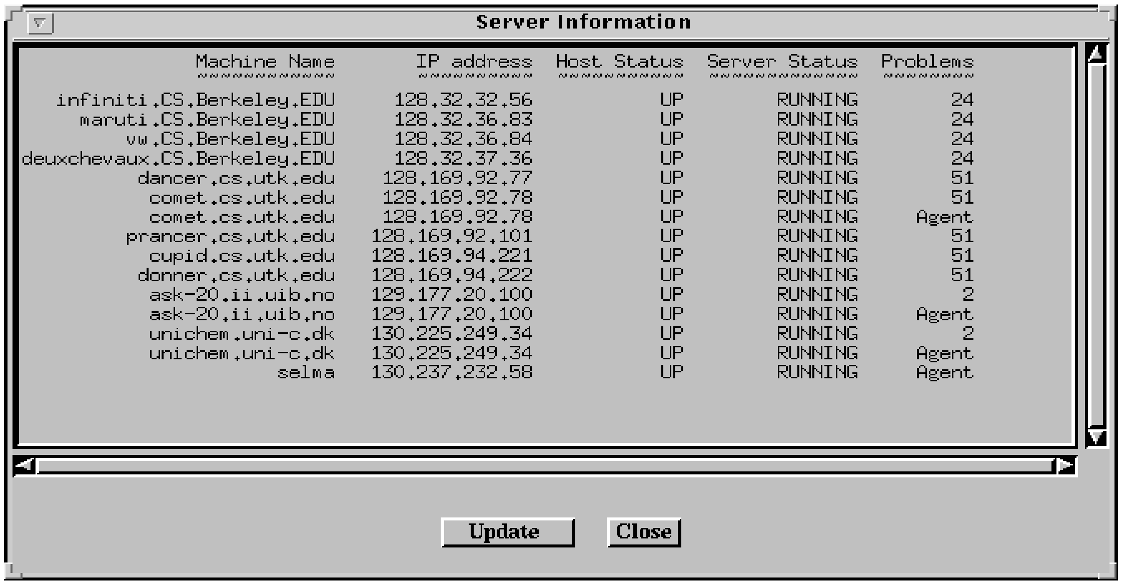


Figure 2.3: The Output Screen

The *Output Window* is arranged like the *Data Input Window*, with a list of objects on the left, a data box on the right, and a description box on the bottom. When the user clicks on any item in the *Output List*, the *Output Description Box* is updated with text describing that object and the *Data Box* is updated with the results of the computation. Above the *Data Box* is a "Save" button which allows users of the stand-alone application to save the text in the *Data Box* to a file. Note that the data saved is that for the selected output object only, not all output objects.
Like the *Data Input Window*, the title bar of the *Output Window* also contains the problem name and a *Request Number*. However, the *Request Number* is slightly different in this window. It consists of two numbers separated by a "." (period). The first number is the *Request Number* from the *Data Input Window* from which this output originated. The second number uniquely identifies this window so that it can be distinguished from other *Output Windows*. Here's an example of how the numbers are assigned: the user chooses a problem, "ddot" perhaps, on the initial screen and clicks "Solve". The *Data Input Window* corresponding to that problem will have *Request Number* "1". Then the user chooses a different problem, "matmul" perhaps, and clicks "Solve". The *Request Number* corresponding to that problem will be "2". The number is incremented each time a new input window is opened. The user enters data into the "matmul" window and clicks "Compute" three times to solve three instances of that problem. Soon three output windows will appear with *Request Numbers* "2.1", "2.2", and "2.3" corresponding to the first, second, and third instance of the problem, respectively.

2.6 The Server Information Screen

The *Server Information Screen* allows the user to view the current configuration of the NetSolve system. As shown in Figure 2.4, the *Server Information Screen* consists of a `TextArea` and two buttons. The server information displayed in the `TextArea` may be updated at any time by clicking on the "Update" button. The other button is labeled "Close" and clicking on it makes the window disappear.



```
                Machine Name         IP address   Host Status   Server Status   Problems
                ~~~~~~~~~~~~         ~~~~~~~~~~   ~~~~~~~~~~~   ~~~~~~~~~~~~~   ~~~~~~~~
        infiniti.CS.Berkeley.EDU    128.32.32.56            UP         RUNNING         24
          maruti.CS.Berkeley.EDU    128.32.36.83            UP         RUNNING         24
              vw.CS.Berkeley.EDU    128.32.36.84            UP         RUNNING         24
     deuxchevaux.CS.Berkeley.EDU    128.32.37.36            UP         RUNNING         24
                dancer.cs.utk.edu  128.169.92.77            UP         RUNNING         51
                 comet.cs.utk.edu  128.169.92.78            UP         RUNNING         51
                 comet.cs.utk.edu  128.169.92.78            UP         RUNNING      Agent
               prancer.cs.utk.edu 128.169.92.101            UP         RUNNING         51
                 cupid.cs.utk.edu 128.169.94.221            UP         RUNNING         51
                donner.cs.utk.edu 128.169.94.222            UP         RUNNING         51
                 ask-20.ii.uib.no 129.177.20.100            UP         RUNNING          2
                 ask-20.ii.uib.no 129.177.20.100            UP         RUNNING      Agent
                 unichem.uni-c.dk 130.225.249.34            UP         RUNNING          2
                 unichem.uni-c.dk 130.225.249.34            UP         RUNNING      Agent
                            selma 130.237.232.58            UP         RUNNING      Agent
```

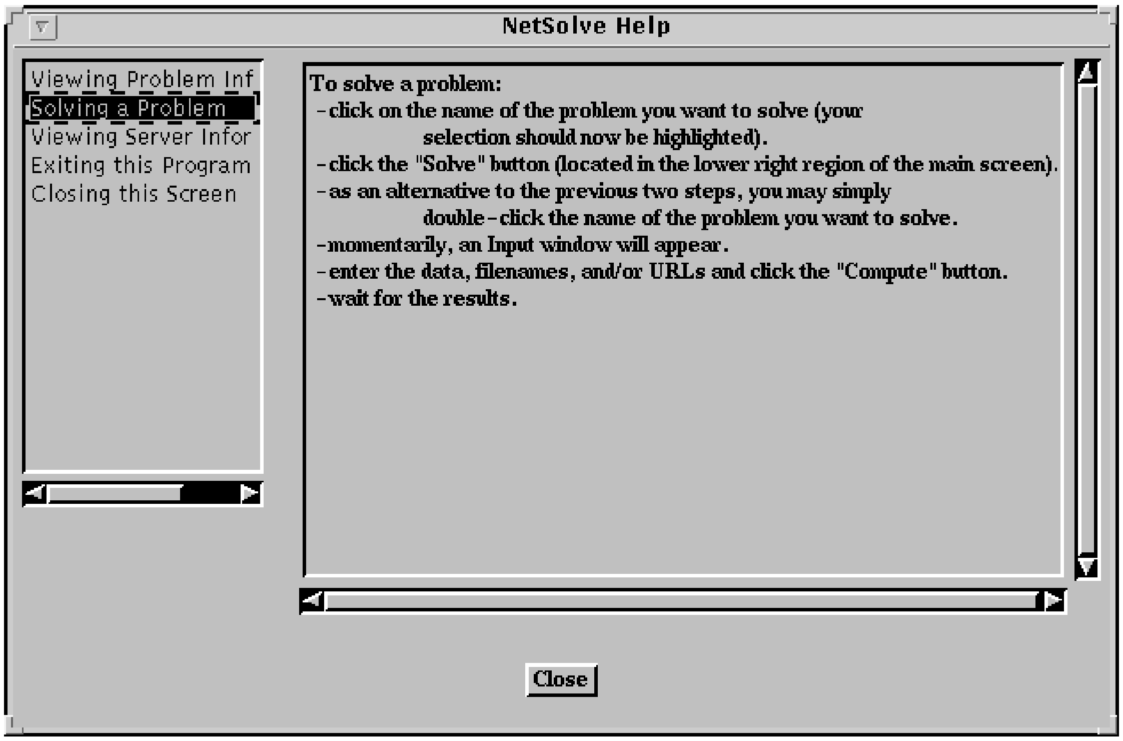
Figure 2.4: The Server Information Screen

2.7 The Help Screen

The *Help Screen*, as shown in Figure 2.5, is designed to provide users with information about using the graphical interface. On the left, a `List` contains several topics that the user may choose from by clicking on the list item. On the right, a `TextArea` displays helpful information on the topics, which range from viewing problem information to quitting the program. Since the *Help Screen* does not interfere with the operation of the main screen, the user may keep both screens open simultaneously. This allows the user to read the help text and then perform the suggested actions without having to close either window. The *Help Screen* may be dismissed when no longer needed by clicking on the "Close" button.

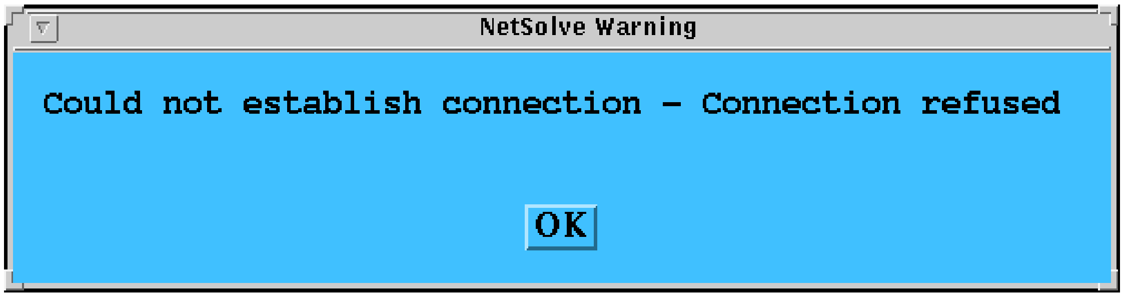
2.8 Error Screens

There are two types of error screens used in the Java interface. The first, as shown in Figure 2.6, is designed to display very short, but important, messages. To help capture the user's attention, the window background uses a prominent color and the text is displayed in a large font. Further, to ensure that the user acknowledges the error message, the parent window is disabled until the user clicks the "OK" button in the error message window. The second type of error screen is designed to display much more text than the first type, so it contains a `TextArea` with a smaller font. As shown in Figure 2.7, this screen is primarily used to provide users a report of the errors found in their data. The second type of error screen does not disable the parent window, though, because it helps the user correct problems with the input data if the error messages can be displayed in one window while the input data is edited in the *Data Input Window*.



To solve a problem:
-click on the name of the problem you want to solve (your
selection should now be highlighted).
-click the "Solve" button (located in the lower right region of the main screen).
-as an alternative to the previous two steps, you may simply
double-click the name of the problem you want to solve.
-momentarily, an Input window will appear.
-enter the data, filenames, and/or URLs and click the "Compute" button.
-wait for the results.

Figure 2.5: The Help Screen



Could not establish connection – Connection refused

Figure 2.6: The Error Dialog Box

Figure 2.7: The Error Report Screen

Chapter 3

The NetSolve Agent and Server

3.1 Getting Started

3.1.1 Downloading the Software

The server software can be downloaded from the NetSolve homepage at

`http://www.cs.utk.edu/netsolve/server_distribution.tar.gz`.

The following UNIX commands will create the `Netsolve_server` directory:

```
% gunzip server_distribution.tar.gz
% tar -xvf server_distribution.tar
```

3.1.2 Setting the Architecture

The `Netsolve_server` directory includes 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
```

or preferably, if `netsolvegetarch` is in the path:

```
setenv NETSOLVE_ARCH `netsolvegetarch`
```

To date, NetSolve has been ported to the following different architectures:

- 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 IRIS running OS $\geq$ 5.0
- SGI64 : Silicon Graphics IRIS running OS $\geq$ 6.0
- HPPA: HP 9000 PA-Risc
- RS6K: IBM RS/6000

3.1.3 Setting the root

In addition to the `NETSOLVE_ARCH` environment variable it is necessary to set an environment variable that contains the path to the `Netsolve_server` directory. It should be defined in the `.cshrc` as:

```
setenv NETSOLVE_SERVER_ROOT /home/me/Netsolve_server
```

for instance.

3.1.4 Compiling

Now that the `NETSOLVE_ARCH` and `NETSOLVE_SERVER_ROOT` environment variables have been set as described in Section 3.1.2, the software can be compiled. First, one should go to the `Netsolve_server/conf` directory and edit the `$NETSOLVE_ARCH.def` file (for instance `RS6K.def`). This file contains a custom section in which the user can modify the compilation parameters. Here is a typical section:

```
# ---- Custom Section ----
F77             = f77
CC              = cc
LINKER = f77
# ---- End of Custom Section ----
```

This custom section specifies which compilers are going to be used. These parameters can be modified before compilation. However, the file also contains other information that should not be modified. Typing `make` in the `Netsolve_server` directory will give instructions to complete the compilation.

3.2 The Agent

The executable of the NetSolve agent is located in:

```
$NETSOLVE_SERVER_ROOT/bin/$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 comet.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.

3.3 The Server

3.3.1 Starting a Server

The executable of the NetSolve server is located in:

`$NETSOLVE_SERVER_ROOT/bin/$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_SERVER_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 a server configuration file.

3.3.2 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 `<hosname>`. Otherwise, the server will only be known to the agent on `<hosname>`.

- `'@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.

- `'@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 [5, 6] 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 3.5.

- `'@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_SERVER_ROOT/server_config` should be used as a template to create new configuration files.

3.3.3 Customizing the Server

The default server configuration file contains several lines that specify description files. These files are located in the `$NETSOLVE_SERVER_ROOT/problems` directory. This directory contains many description files that are not used by the default NetSolve server. These files correspond to problems that are solved with numerical software that is not distributed with NetSolve. If these files are to be used, one must add the corresponding lines to the configuration file. It is also necessary to update the file `$NETSOLVE_SERVER_ROOT/conf/$NETSOLVE_ARCH.def` by adding to the variable `NUMLIB` the path to the required numerical libraries. The server can then be re-compiled by typing

```
% make server
```

in the `$NETSOLVE_SERVER_ROOT` directory. Section 3.5 gives details on the creation of new problem description files.

3.4 Managing the System

It may become difficult to keep track of the agents and servers that take part in a NetSolve system. It is always necessary to know the name of an host running one agent in the system to find out information on that system. It is then possible to use the CGI scripts on the NetSolve homepage to obtain a list of participating hosts. The directory `$NETSOLVE_SERVER_ROOT/bin/$NETSOLVE_ARCH` contains two executables called `destroy_agent` and `destroy_server` that can both be called with a hostname as an argument to terminate a NetSolve agent or server on that host. It also contains the two executables `is_there_agent` and `is_there_server` that tak a hostname as argument and print out the agent/server status on the corresponding host.

3.5 Expanding the Server

As already indicated in Section 3.3.3, it is possible to add new functionalities to a NetSolve computation server by specifying additional description files in the server configuration file. Some description files are located in the directory `$NETSOLVE_SERVER_ROOT/problems`. In what follows we describe how a description file can be created. It is strongly advised to use the existing files as templates. The rationale behind everything that is explained in what follows comes from [7]. Each description file is composed of several *problem descriptions*. Before explaining how to create a problem description, we define the concept of *mnemonics*.

3.5.1 Mnemonics

We have already seen that a NetSolve problem takes some objects in input and produces some objects as output. Generally, the objects are scalars, vectors or matrices of Fortran data-types. 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 and vectors),
- Number of columns : `n[I|O]`x (only for matrices),
- Leading dimensions : `l[I|O]`x (only for matrices).

For example, `mI4` designates the number of rows of the input object number 4, whereas `O1` designates the pointer to the element(s) 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.

3.5.2 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 <name>'` 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 3.5.4). There can be several such lines as a problem can call several functions.
- `'@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 [4], the major has to be `'COL'`.
- `'@PATH <path>'` specifies a URL-like name for the problems. This path is only a naming convention and is used for presentation purposes for interactive interfaces (Matlab and the Java GUI).
- `'@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>'` specifies an object type and data type. This line is followed by a textual description of the object. The currently available object types are:
  - `MATRIX`,
  - `VECTOR`,
  - `SCALAR`,
  - `UPF` (User-provided function (see Section 4).

  and the possible data types are

  - `CHAR` : character,
  - `B` : byte (character that is never XDR encoded),
  - `I` : integer,
  - `S` : single precision real,
  - `D` : double precision real,
  - `C` : single precision complex,
  - `Z` : double precision complex,
  - `EXTERNAL` : User-provided function (see Section 4).

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>'` specifies an object type (scalar, vector and matrix) and data type (one of the Fortran data type). This line is followed by a 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 2.2.2).
- `'@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

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

3.5.3 Calling Sequence

The calling sequence for 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 or Java. Indeed, when using a *high-level* interface like Matlab, the objects in input and output to the problem are self-contained. This means that a single identifier described entirely the object. From C and Fortran, such a data encapsulation is not possible and the calling sequence needs then to be precisely defined. 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`.

- `'@FORMATS <number>'` specifies how many different calling sequences are available for the problem. this feature is not activated yet in the current version of the software and this line should **always** be:

  ```
  @FORMATS 1
  ```

  `'@FORMAT'` 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>=<number>'` 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.

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, in fact, C code that uses the mnemonics described in Section 3.5.1. 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 integer that is number of rows of the first object in input.
- `'@nO1@'`: **pointer** to 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 (for instance) of matrix `a`. This can be done as follows:

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

The macro `BAD_DIMENSION` is defined by NetSolve. Other macros available are `BAD_VALUES` (for invalid input parameters), `FAILURE` (for a malfunction of the numerical software) or `NO_SOLUTION` (sometimes useful if no numerical solution has been found). 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. 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.

3.5.4 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"
@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
Matrix A (NxN)
@OBJECT MATRIX D
Matrix B (NxNRHS)
```

```
@OUTPUT 1
@OBJECT MATRIX D
Solution X (NxNRHS)
@COMPLEXITY 3,3
@FORMATS 1
@FORMAT
@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
```

3.5.5 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.

Chapter 4

The User-Supplied Function Feature

4.1 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*.

4.2 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.

4.3 For the Client

4.3.1 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 1.2.3). 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.

4.3.2 From MATLAB

From MATLAB, when the user consults the list of available problems, he can determine whether any given problem requires one or more user-supplied functions. If the problem does indeed require such functions, then these functions have to be written in files. These files have to be in the working directory and their names are passed to the call to NetSolve. The problem is then called as described in Section 2.2. If something is wrong with the user-supplied function, `netsolve()` and `netsolve_nb()` print out special error messages.

4.3.3 From C or Fortran

The situation from C or Fortran is almost the same as from MATLAB. The user-supplied functions have to be in files in the UPF working directory. However, we introduce here a new function, called `netsldir()`, that sets the default directory in which to look for the function file. The names of the files are then passed to the call to NetSolve. A typical call to `netsldir()` in C is

```
netsldir("/homes/me/my_functions");
```

and in Fortran is

```
      NETSLDIR('/homes/me/my_functions')
```

Here, `netsl()` and `netsldir()` return special NetSolve status codes concerning the user-supplied function.

4.3.4 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.

4.3.5 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).

4.4 For the Server

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

```
@OBJECT UPF EXTERNAL
```

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 close to impossible on some platforms.

4.5 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.

>> `[ ... ] = 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`.

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 2.3.2 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 2.3.2 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()`.

`netsldir("`*<directory name>*`")`

Sets the default directory where the user-supplied functions are located.

`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()>*`()',NSINFO, ...)`

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 2.3.2 for a discussion about this calling sequence. When the call returns, the integer `NSINFO` 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()>*`()',NSINFO, ...)`

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 2.3.2 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>*`,NSINFO)`

Waits for a request completion. `FNETSLWT()` takes as argument a request handler and an integer. When the call returns, NSINFO 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>*`,NSINFO)`

Probes for a request completion. `FNETSLPR()` takes as argument a request handler and an integer. When the call returns, NSINFO 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 FNETSLDIR('<*directory name*>')

Sets the default directory where the user-supplied functions are located.

CALL FFNETSLERR(<*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
`NetSolveSuccess`	1	Successful call to a routine
`NetSolveNotReady`	0	Request not yet completed
`NetSolveFailure`	-1	Failure of the NetSolve system
`NetSolveBadCode`	-2	Badly formatted problem name
`NetSolveUnknownProblem`	-3	Unknown problem in the system
`NetSolveBadInput`	-4	Wrong number/type of input
`NetSolveBadOutput`	-5	Wrong number/type of output
`NetSolveAgentFailure`	-6	Failure of the NetSolve Agent
`NetSolveNoServers`	-7	No computational resource available
`NetSolveBadDimension`	-8	Incorrect dimensions of non scalar input data
`NetSolveNoSolution`	-9	No solution for this problem given the input data
`NetSolveRequestFull`	-10	No more requests possible
`NetSolveInvalidRequestNumber`	-11	Unknown request handler
`NetSolveSetArch`	-12	Environment variable NETSOLVE_ARCH should be set
`NetSolveSetAgent`	-13	Environment variable NETSOLVE_AGENT should be set
`NetSolveNoAgent`	-14	No agent running on $NETSOLVE_AGENT
`NetSolveBadValues`	-15	Incorrect numerical values of the input
`NetSolveFileNotFound`	-16	No file containing a user-supplied function
`NetSolveFileReadError`	-17	Impossible to read file containing the user-supplied function
`NetSolveUPFFailed`	-18	Compilation error of the user-provided function
`NetSolveUPFUnsafe`	-19	Unsafe user-provided function
`NetSolveBadCallingSequence`	-20	Erroneous calling sequence

Appendix E

NetSolve Java API Reference

For current documentation of the entire NetSolve API, please consult the following Web page:

`http://www.cs.utk.edu/netsolve/JavaAPI/index.html`

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

```
/**
* DgesvExample
*
* This class illustrates a simple example of using the
* NetSolve Java API.  Two linear systems are solved (dgesv)
* using blocking calls.  Then the results are printed out.
*
* @version 1.0, 22 Jan 1998
* @author Keith Seymour (seymour@cs.utk.edu)
*
* @see Netsolve
* @see NetsolveSession
**/

import java.util.Vector;

public class DgesvExample extends Example {
  public static void main (String [] args) {
    DgesvExample mm = new DgesvExample();
    NetsolveSession ns;
    Netsolve n;
    Netsolve n2;
    double [][] a1 = {     // Initialize the data
      {1.0,   2.0,  3.0,  4.0},
      {5.0,   6.0,  7.0,  8.0},
      {9.0,  10.0, 11.0, 12.0},
      {13.0, 14.0, 15.0, 16.0}
    };
    double [][] a2 = {     // Initialize the data
      {1.0,   2.0,  3.0,  4.0},
      {13.0, 14.0, 15.0, 16.0},
      {9.0,  10.0, 11.0, 12.0},
      {5.0,   6.0,  7.0,  8.0}
    };
    Vector out;
    Integer outInt;
    double [][] outMat1, outMat2;
    int [] outVec;
```

```
// Since these variables are all set within the 'try'
// block, the compiler can't confirm that they are
// initialized.  To avoid the subsequent complaints from
// the compiler, we initialize them to null here.

n = null;
n2 = null;
ns = null;
out = null;

// Almost all of the methods in Netsolve and NetsolveSession
// throw NetSolveException upon some error condition, so it's
// easier to just enclose the whole transaction in a try-catch
// block than to try each one individually (although that is
// possible, too).

try {
  ns = new NetsolveSession("woodstock.cs.utk.edu");

  n = new Netsolve(ns);     // Specify 'session'
  n.setProblem("dgesv");    // Specify the problem to solve
  n.pushArg(a1);            // Pass first parameter to dgesv
  n.pushArg(a1);            // Pass second parameter to dgesv
  n.submitProb();           // Submit this problem
  out = n.getOutput();      // Get the output item(s)
}catch(NetSolveException e) {
  System.err.println("Error in first submission:");
  System.err.println(e.getMessage());
  System.exit(1);
}

// Print results from first submission.

System.out.println("Results from first submission:");

outMat1 = (double [][]) (out.elementAt(0));
outVec  =      (int []) (out.elementAt(1));
outMat2 = (double [][]) (out.elementAt(2));
outInt  =     (Integer) (out.elementAt(3));

mm.printMat( outMat1 );
mm.printVec( outVec );
mm.printMat( outMat2 );
System.out.println( outInt.intValue() );

// The second submission is enclosed in a separate try-catch
// so that we may distinguish between error messages generated
// during the first and second submissions.  Notice that in
// initializing this Netsolve object, we reuse the NetsolveSession
// from the previous submission.  This saves time by only
// retrieving the problem list once.  However, if you wanted
// to submit the second problem to a different agent, you
// would have to instantiate a new NetsolveSession, specifying
// that agent.
```

```
    try {
      n2 = new Netsolve(ns);     // Specify 'session'
      n2.setProblem("dgesv");    // Specify the problem to solve
      n2.pushArg(a1);            // Pass first parameter to dgesv
      n2.pushArg(a2);            // Pass second parameter to dgesv
      n2.submitProb();           // Submit this problem
      out = n2.getOutput();      // Get the output item(s)
    }catch(NetSolveException e) {
      System.err.println("Error in second submission:");
      System.err.println(e.getMessage());
      System.exit(1);
    }

    // Print results from first submission.
    System.out.println("Results from second submission:");

    outMat1 = (double [][]) (out.elementAt(0));
    outVec  =      (int []) (out.elementAt(1));
    outMat2 = (double [][]) (out.elementAt(2));
    outInt  =     (Integer) (out.elementAt(3));

    mm.printMat( outMat1 );
    mm.printVec( outVec );
    mm.printMat( outMat2 );
    System.out.println( outInt.intValue() );

  }
}
```

Bibliography

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

[2] H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[3] Inc The Math Works. *MATLAB External Interface Guide.* The Math Works, Inc, 1992.

[4] 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.* SIAM Philadelphia, Pennsylvania, 2 edition, 1995.

[5] 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.

[6] 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.

[7] H. Casanova and J. Dongarra. Providing uniform access to numerical software. In *IMA Volume on Algorithms for Parallel Processing.* To appear in proceedings, 1996.