Client User's Guide
to
NetSolve

Henri Casanova[*] Jack Dongarra[*][†] Keith Seymour[*]

December 6, 1996

Abstract

The NetSolve system, developed at the University of Tennessee, is a client-server application designed to solve computational science problems over a network. Users may access NetSolve computational servers through C, Fortran, MATLAB, or Java interfaces. This document briefly presents the basics of the system. It then describes in detail how the different clients can contact the NetSolve system to have some computation performed, thanks to numerous examples. Complete reference manuals are given in the appendixes.

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

Contents

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. The purpose of this document is to describe all those interfaces and the way they interact with the NetSolve servers.

The next section gives basic information about the NetSolve system. A complete description of the software layout and communication protocols can be found in [2]. Section 3 provides all the information needed to download and install NetSolve. Sections 4, 5, and 6 describe in detail all the interfaces. Section 7 describes a recently developed feature of NetSolve, the "user-supplied function" mechanism.

2 Overview of the NetSolve System

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. Moreover, the NetSolve system can be running in a *heterogeneous* environment, which means that machines with different data formats can be in the system at the same time.

Figure 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.

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].

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

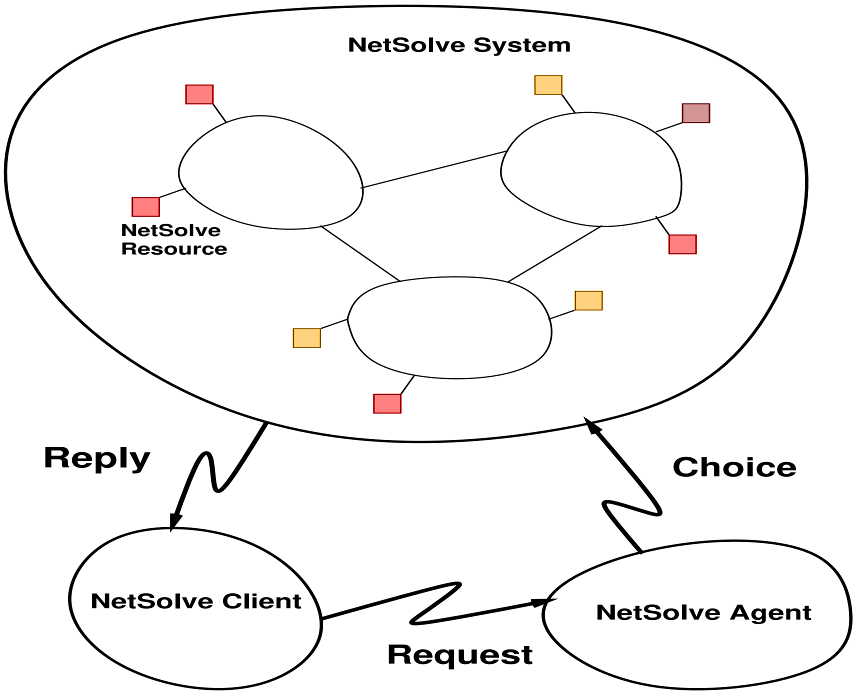

Figure 1: The NetSolve System

- *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 C and Fortran interfaces are usually designed so that they fit the underlying scientific software calling sequence.

2.3 Problems Solvable 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 can be useful, as seen in Section 5.2.

3 Getting Started

3.1 Downloading and Installing 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
```

The different interfaces can now be compiled.

3.2 Setting the Architecture

The `Netsolve_client` directory includes a 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, 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 ¿= 5.0
- HPPA: HP 9000 PA-Risc
- RS6K: IBM RS/6000

3.3 Setting an Agent Name

As described in Section 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
```

3.4 Compiling the Client

Now that the `NETSOLVE_ARCH` environment variable has been set as described in 3.2, 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
# ---- 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. 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.

4 MATLAB Interface

4.1 Introduction

Building the MATLAB interface as described in 3.4 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 `.mex4` for SPARCs). 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.

4.2 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 NetSolve system available.
As stated briefly in Section 2.3, it is possible to obtain the list of solvable problems from MATLAB. Let us try that first:

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

Every line contains a full problem name. This list can be really long, and in that case it is wiser to use the CGI scripts in Section 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 NetSolve 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 3.3). 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 that contains 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.
In the next section, we will see how to solve a problem.

4.3 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 4.4.
Let us go on with the `eig` example we started to develop in the preceding section. The user now knows that he has to provide a double-precision square matrix to NetSolve, and he knows that he is going to get two real vectors back (or one single complex vector). He first creates a $300 * 300$ matrix, for instance,

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

The call to NetSolve is now

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

All the calls to `netsolve()` will look the same. The left-hand side must contain the output arguments, in the same order as listed in the *output description* (see Section 4.2). The first argument to `netsolve()` is always the name of the problem. After this first argument the input arguments are listed, in the same order as they are listed in the *input description* (see Section 4.2). 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 any problem. To merge x and y, the user 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
        .........
        .........
```

4.4 Calling `netsolve_nb()`

The obvious drawback of the function `netsolve()` is 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 in NetSolve, 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 4.3, 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 right-hand side of `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 callas `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  : 2324 seconds
Request #1 - eig
        Assigned to vw.cs.Berkeley.edu 3 seconds ago
        Still RUNNING
        Predicted execution time  : 2606 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.

4.5 What Can Go Wrong?

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

4.5.1 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.

4.5.2 User Mistakes

The second class of error comes from the user. If the user does not follow the calling sequences described in Sections 4.3 and 4.4, 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.

5 C and Fortran Interfaces

5.1 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 4 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 3.4, 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 these files to either one of these libraries to create a C or Fortran program calling NetSolve. The user must also include the following header file:

- `Netsolve_client/include/netsolve.h`

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

5.2 Knowing the Calling Sequence

When we described the MATLAB interface in Section 4, 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 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 of Fortran is to preserve the native calling sequences of the numerical software. Recall that in Section 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. Thus, 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.

5.3 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 5.2 and their number of variables. The C prototype of the function is

```
int netsl(char *problem_name, ... < calling sequence > ...)
```

and the Fortran prototype is

```
SUBROUTINE NETSL(PROBLEM_NAME, NSINFO, ... < calling sequence > ...)
```

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 NETSL('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*. 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. Becauses we conserved 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!
describes `netslnb()`, the nonblocking version.

5.4 Nonblocking Call

We developed this nonblocking call for the same reason we developed one for MATLAB (see Section 4.4): 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 NETSLNB('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 `netslpb()`. If the call is successful, the function returns immediately with either an 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 NETSLPB(REQUEST,NSINFO)
```

and in C :

```
nsinfo = netslpb(request);
```

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

```
call NETSLWT(REQUEST,NSINFO)
```

and the C call :

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

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

5.5 Error messages

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

```
netslerr(nsinfo);
```

and in Fortran

```
        call NETSLERR(NSINFO)
```

5.6 Built-in Examples

C and Fortran 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 E and F.

6 Java Interface

6.1 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 restriction to the capabilities of applets. At this time, it appears to be impossible to open sockets to a remote hosts, making the NetSolve interface unusable. Future versions of these Web Browsers will undoubtedly alleviate these problems.
To start the stand-alone application:

```
java NetSolveClient blah.cs.utk.edu
```

where blah.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.

6.2 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 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.

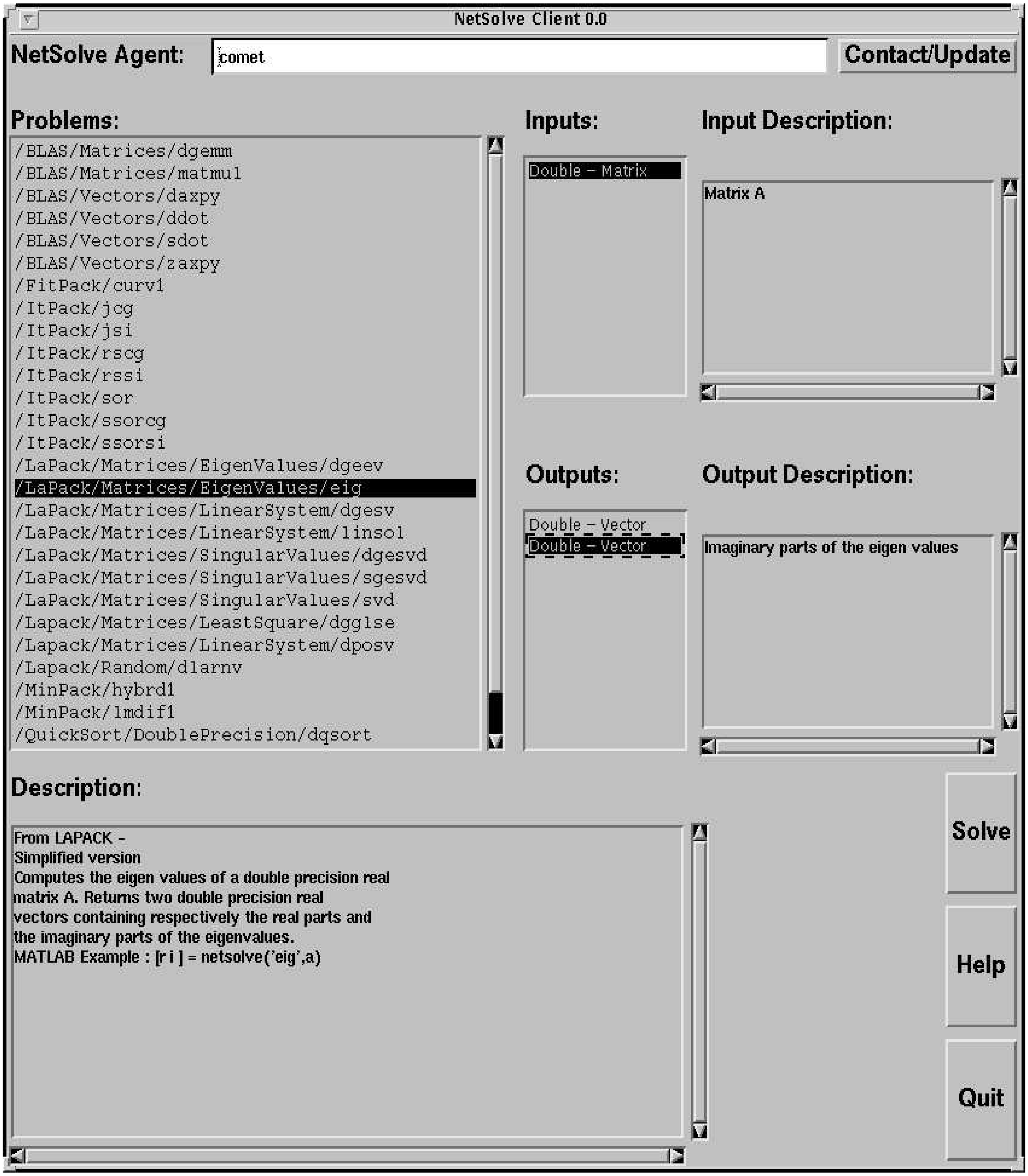

Figure 2: The Initial Screen

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.

6.3 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. A new window will appear allowing the user to input data for each input object required by the problem. Figure 3 shows the *Data Input Window*, which consists of the following components:

- Input List
- Input Description Box
- Filename (or URL) Selection Box
- Data Input 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 to specify the name of a file containing the data in the *Filename Selection Box*. Next to the *Filename Selection Box* is a "Browse" button which allows choosing the file using a graphical file browser. Those users accessing the NetSolveClient via a Web browser will have a *URL Selection Box* (instead of a *File Selection Box*) in which they may type in the URL for their data file. This allows NetSolve to access the user's local data files over the network. 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 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 6.4) 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:

- 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.

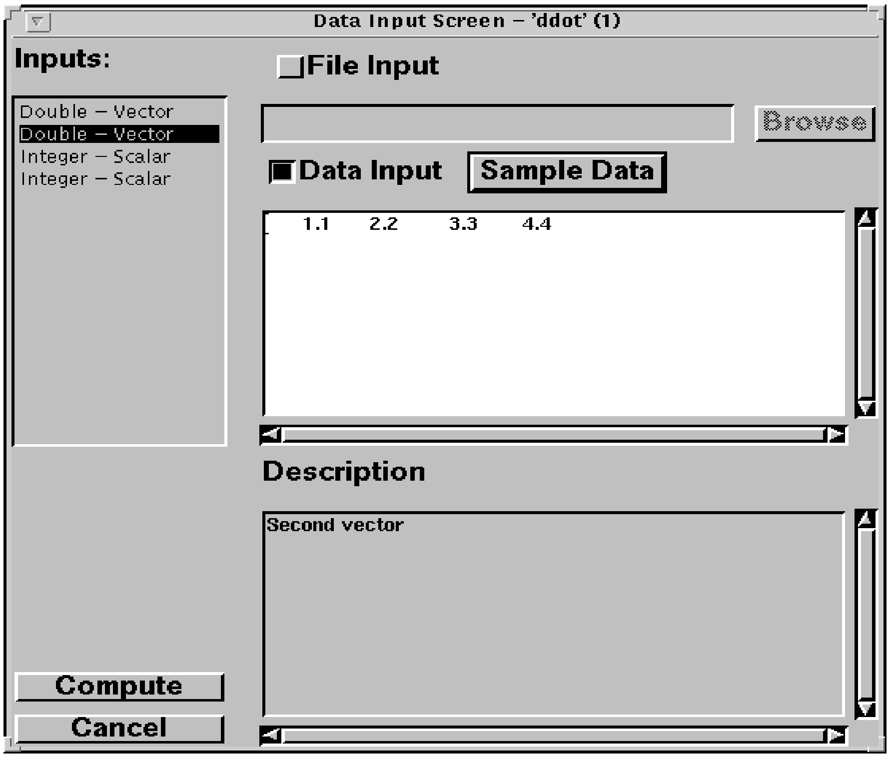

Figure 3: The Input Screen

- 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.

6.4 Viewing the Results

Once the computational server sends back the results, a new window appears allowing the user to browse the results. Figure 4 shows the *Output Window*, which consists of the following components:

- Output List
- Output Description Box
- Data Box

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

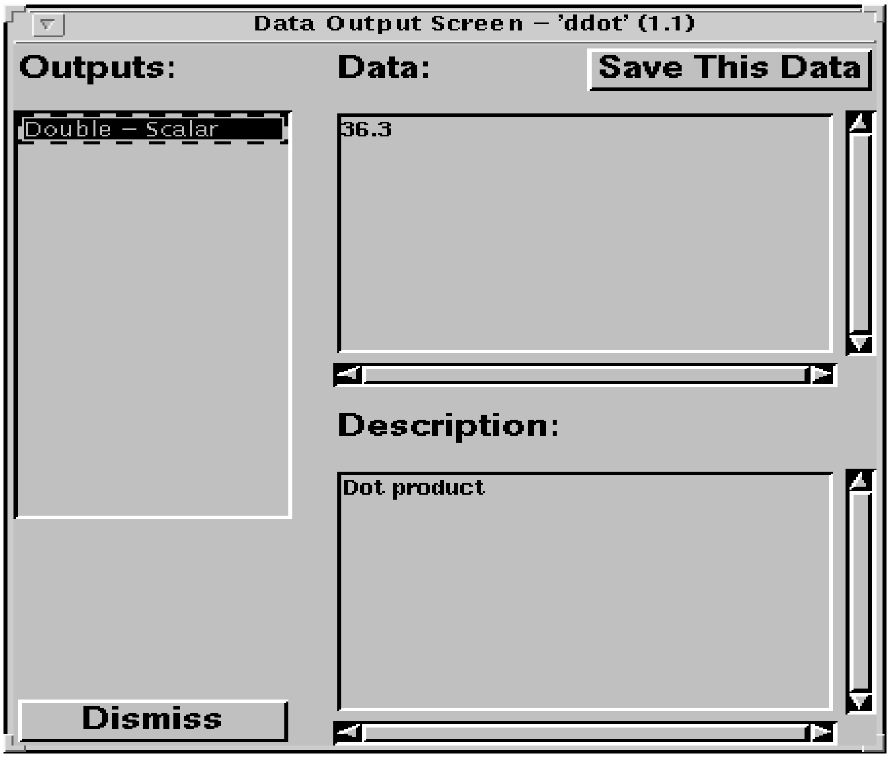

Figure 4: The Output Screen

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.

7 The User-Supplied Function Feature

7.1 Motivation

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

7.2 Solution

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

7.3 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 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 software require the user to provide more than one function. When that is the case, the description of the problem mentions it and give all the prototypes for all the functions to supply.

7.4 From MATLAB

From MATLAB, when the user consults the list of available problems, he can determine whether any given problem requires a user-supplied function. If the problem does indeed require such a function, this function has to be written in a file. This file can be called `upf.f` or `upf.c`, depending on the language used to write it. This file has to be in the current working directory. The problem is then called as described in Section 4. If something is wrong with the user-supplied function, `netsolve()` and `netsolve_nb()` print out special error messages.

7.5 From C or Fortran

The situation from C or Fortran is almost the same as from MATLAB. The user-supplied function has to be in `upf.f` or `upf.c`, in the working directory. However, we introduce here a new function, called `netsldir()`, that sets the default directory in which to look for the function file. 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.

7.6 From Java

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

7.7 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 used-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.

8 General Conclusion

NetSolve is a new project, and as such is bound to undergo a lot of changes in a very close future. However, we believe that all the general ideas presented in this document about problem specification, as well as the details of each interface will not be highly modified. The changes in NetSolve should be in fact be focused on its way of operating internally and should not have any impact on the interfaces. Of course, some new features are going to be added along the way, and they will surely be described in a next version of this User's Guide.

One of our goal in designing the different clients to NetSolve was to keep them as straightforward as possible. This can be seen best with the MATLAB and Java interfaces. The C and FORTRAN interfaces are more complicated because those languages, unlike MATLAB, do not provide a high level of abstraction. However, we emphasized that those two interfaces allow users of numerical software to *switch* very easily to the NetSolve counterpart of those softwares. The efficiency of the computation is outside the scope of this document and is discussed in great detail in [2].

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 request's 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's 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`.

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 5.2 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 **nonbloking** request to NetSolve. `netslnb()` takes as argument the name of the problem, and the list of arguments in the calling sequence. See Section 5.2 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's 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 = netslpb(<request handler>)`

Probes for a request's completion. `netslpb()` 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("<problem name>")`

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

`netslerr("<error code>")`

Displays an explicit error message given a NetSolve error code.

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 NETSL('<*problem name*>',NSINFO, ...)

Sends a **blocking** request to NetSolve. `NETSL()` takes as argument the name of the problem, an integer, and the list of arguments in the calling sequence. See Section 5.2 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 NETSLNB('<*problem name*>',NSINFO, ...)

Sends a **nonbloking** request to NetSolve. `NETSLNB()` takes as argument the name of the problem, an integer, and the list of arguments in the calling sequence. See Section 5.2 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 NETSLWT(<*request handler*>,NSINFO)

Waits for a request's completion. `NETSLWT()` 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 `NETSLNB()`.

CALL NETSLPB(<*request handler*>,NSINFO)

Probes for a request's completion. `NETSLPB()` 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 `NETSLNB()`.

CALL NETSLDIR('<*problem name*>')

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

CALL NETSLERR("<*error code*>")

Displays an explicit error message given a NetSolve error code.

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 nonscalar 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

E 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;
  }

   /* 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);
  }
}
```

F Complete Fortran Example

```
#include "netsolve.h"
*******************************************
*                                         *
* TEST of the FORTRAN INTERFACE to NETSOLVE*
*                                         *
*******************************************

      PROGRAM TEST

      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 NETSL('Eig',INFO_NS,
     $                A,MAX,MAX,X1,Y1)

      IF(INFO.LT.0) THEN
       WRITE(*,*) INFO
       STOP
      ENDIF

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

      IF(REQUEST.LT.0) THEN
       WRITE(*,*) REQUEST
       STOP
      ENDIF

      CALL NETSLWT(REQUEST,INFO_NS)
      IF(INFO.LT.0) THEN
       WRITE(*,*) INFO
       STOP
      ENDIF

      STOP
      END
```

References

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

[2] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. In *Supercomputing '96, Pittsburgh*. To appear in proceedings, 1996.

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