Kestrel: An Interface from Optimization Modeling Systems to the NEOS Server

Elizabeth D. Dolan

Industrial Engineering and Management Sciences Department, Northwestern University
Mathematics and Computer Science Division, Argonne National Laboratory

dolan@mcs.anl.gov

Robert Fourer

Industrial Engineering and Management Sciences Department, Northwestern University

4er@iems.northwestern.edu

Jean-Pierre Goux

Artelys S.A.

jean-pierre.goux@artelys.com

Todd S. Munson

Mathematics and Computer Science Division, Argonne National Laboratory

tmunson@mcs.anl.gov

Jason Sarich

Industrial Engineering and Management Sciences Department, Northwestern University
Mathematics and Computer Science Division, Argonne National Laboratory

sarich@mcs.anl.gov

September 2002; revised December 2004, December 2006

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Science Foundation Information Technology Research initiative under grant CCR-0082807; and by the National Science Foundation Operations Research program under grant DMI-0322580.

Abstract. The NEOS Server provides access to a variety of optimization resources via the Internet. The new Kestrel interface to the Server enables local modeling environments to request NEOS optimization services and retrieve the results for local visualization and analysis, so that users have the same convenient access to remote NEOS solvers as to those installed locally. Kestrel agents have been implemented for the AMPL and GAMS modeling environments; these agents have been designed so that subproblems can be queued for execution and later retrieval of results, making possible a rudimentary form of parallel processing.

1. Introduction

The NEOS Server [10, 11, 23], `www-neos.mcs.anl.gov`, provides access through the Internet to over 50 solvers for a range of optimization problem types. This paper describes the design and creation of Kestrel, a callable interface to the NEOS Server that extends the Server's versatility by permitting access through a broad variety of modeling languages and systems.

The Kestrel interface is called from a locally running environment, and returns results to that environment. Kestrel thus gives a modeling system the same access to remote NEOS solvers as to solvers installed locally. As a result, analysts can consider a wider variety of solvers, problems may be solved more effectively, and new solver technologies may be disseminated more rapidly.

Section 2 reviews the aspects and limitations of the NEOS Server that have led to our design of Kestrel. Section 3 then presents Kestrel from a technical point of view. It considers the communication and interface problems inherent in calling the NEOS Server from a program somewhere else on the Internet, together with our solutions to those problems as embodied in the design of the NEOS application programming interface and the Kestrel agents for modeling systems.

The remainder of the paper illustrates how we have designed the Kestrel interface to make it useful from the modeler's point of view. Section 4 shows how Kestrel agents enable the users of two quite different optimization modeling languages—AMPL [17, 18] and GAMS [5, 7]—to invoke NEOS solvers in virtually the same way as locally installed solvers. Section 5 then presents two more sophisticated examples: one that employs Kestrel/AMPL as part of a larger iterative scheme to solve and visualize solutions to a partial differential equation, and one that incorporates the Kestrel interface to make a GAMS-to-AMPL preprocessing service available via the NEOS Server. By using a paradigm similar to that described in [15], an extension to the AMPL and GAMS Kestrel agents enables subproblems to be queued for execution and later retrieval of results; Section 6 describes this and related features and their applications.

2. The NEOS Server

The NEOS Server is the most ambitious realization to date of the optimization server concept [19]. Operated by the Optimization Technology Center of Argonne National Laboratory and Northwestern University, it represents a collaborative effort involving over 40 designers, developers, and administrators around the world. The project solicits solvers for all kinds of local and global optimization and related problems, including linear, integer, nonlinear, stochastic, and semidefinite programs, with linear, nonlinear, complementarity and more general nondifferentiable constraints. Some of the solvers are free or are available through open-source licenses; others are proprietary to varying degrees, but their owners have agreed to permit their use at no charge through the Server.

Since its inception in 1996, the NEOS Server has been progressively enhanced and now regularly handles 10,000–20,000 submissions per month from a variety of business, educational, and scientific endeavors [13, 14]. Aside from the Kestrel interface to be described, problems are sent to the Server via email, web-based forms, and the NEOS Graphical User Interface (the NEOS GUI). Although the Server's

location is fixed, each of the solvers can run on any workstation connected to the Internet. Hence available computing resources have been able to grow along with the number of solvers and users [12].

The large number of optimization problem types and solvers precludes standardization on any one input format. The NEOS Server instead allows optimization solvers to receive whatever text or binary input they are equipped to accept, whether explicit or symbolic or in the form of C or Fortran routines. To cope with this variety, the Server maintains a database that records the characteristics and needs of each solver. A solver's database entry includes the input format it accepts, the remote workstations on which it may be run, user documentation for the various interfaces, and other information used to construct the NEOS Server's web-based form interface. If a solver can be run with different input formats, it must have a database entry for each. With about 50 distinct solvers, the number of database entries is about 120.

Under this arrangement, each submission to the NEOS Server is a single text input file comprising one or more parts delimited by XML tags. (See [33] for a concise introduction to XML.) For email submissions, the user inserts tags explicitly, while the web interface and NEOS GUI insert appropriate tags automatically. Consulting the database entry for the requested solver, the Server unpacks the input's parts into separate files and chooses a remote workstation on which the submission will be run. The files are then transmitted to the workstation, and the remote solver is initiated there.

Compressed input is automatically detected and expanded as part of the unpacking procedure. More complex preprocessing, such as automatic differentiation or modeling language translation, is performed as part of the remote solver activities after the problem has been sent to a workstation.

During the remote solver's execution, all text in the standard output and error streams is passed back to the NEOS Server, and then to the user in the form of a listing returned through the original submission interface. Some modelers are content to look at this listing, while others either write their own processing programs or cut and paste from the listing to other software. But a listing of this sort is problematical when the results are to be manipulated by a program rather than directly by a human modeler, such as when the optimization is part of some larger computational scheme, is embedded in a broader modeling context, or involves proprietary information.

Effective use of an optimization modeling language, in particular, requires that the results be returned in a form appropriate to the associated modeling environment. The human modeler can then access and manipulate the results through the environment's interface, which may provide specialized facilities for browsing data and results, for dealing with multiple files, and for interacting with databases and spreadsheets. These requirements have motivated our interest in an alternative interface that enables a local modeling environment to send an optimization problem to the Server and retrieve the results in much the same way that the environment would interact with a local solver.

The next section describes the technical demands, design, and implementation of a callable interface to NEOS, and the specifics of accessing this interface from algebraic modeling systems via Kestrel. The reader who is mainly interested in

the application of these ideas may want to skip to Sections 4–6, where we describe several uses of Kestrel in the context of the algebraic modeling languages AMPL and GAMS.

3. Building the Kestrel Modeling Language Agents

In the simplest terms, a callable interface to the NEOS Server requires some mechanism for making function calls over the Internet. A local program known as an *agent* can then translate a user request into a remote function call that actually runs on the computer hosting the NEOS Server.

In designing the callable interface to NEOS, we sought to assure that this arrangement would be sufficiently flexible to handle the broad variety of modeling environments and solvers. To that end, we adopted the following design goals:

- ◇ Agents should be implementable on a choice of platforms and in a choice of languages.
- ◇ Agents should be able to issue both blocking and non-blocking calls on the NEOS Server.
- ◇ Agents should be able to send both text and binary data.
- ◇ A broad variety of applications should be implementable with reasonable effort.
- ◇ The calling mechanism should be compatible with the rest of the NEOS Server infrastructure.
- ◇ The Server should be able to handle many simultaneous agent calls.

Fortunately, standard "middleware" is readily available to help meet these goals. Competing technologies include Microsoft's .NET technology (`www.microsoft.com/net`); DCE, the Open Software Foundation's Distributed Computing Environment (`www.opengroup.org/dce`); the Nexus library, a component of the Globus Alliance (`www.globus.org`); the Object Management Group's CORBA (`www.corba.org`); and UserLand Software's XML-RPC (`www.xmlrpc.com`).

The first implementation of Kestrel used the Nexus library, and the second was based on CORBA. Both versions relied on a separate "Kestrel server" running in cooperation with the NEOS Server, and required a shared disk drive for efficiency. When version 5 of the NEOS Server was implemented in 2005, however, a callable interface through XML-RPC was made a central component of its design, and a separate Kestrel-only server was no longer necessary.

Thus we begin this section with a description of the application programming interface (the "NEOS API") that NEOS 5 provides generally through XML-RPC. We next introduce the concept of an agent that calls the NEOS API. The remainder of this section then focuses on the internal design of Kestrel agents for optimization modeling systems. Subsequent sections argue that these agents can provide a variety of valuable services to users of modeling systems.

Fundamentals of the NEOS API. All submissions to the NEOS Server are performed by creating an XML document file to define the input, submitting this file to the Server, and receiving a result file from the Server.

Communication with the NEOS Server is accomplished by use of the XML-RPC (XML Remote Procedure Call) standard [35]. This standard allows an agent process to invoke a specific member function of a remote object by creating an XML document encapsulating the function name and arguments and making a POST request to a knowledgeable `http` server—the same kind of server that handles web page requests. The `http` server parses the XML document, dispatches the arguments to the requested function, forms another XML document encapsulating the results of the function, and returns the latter document as the response to the POST request. Although XML is designed for text, binary data can also be transmitted via XML-RPC by encoding it as `base64` text that is inserted into the XML document.

This mechanism requires a well-defined API to define agent calls that match all of the Server functions. The NEOS API—described at `neos.mcs.anl.gov/neos/NEOS-API.html`—contains over twenty-five calls accessible to C, C++, Java, Python, Perl, PHP, Ruby, and any other language for which an XML-RPC library is available.

As an example, the following Python call submits an XML-delimited optimization request (or "job") to the NEOS Server in the form of a character string:

```
(jobNumber, password) =
   neos.submitJob(xmlstring, user='', interface='', id=0)
```

If the submission fails, such as when the NEOS queue for the requested solver is full, the returned `jobNumber` is 0 and the `password` is an error message. Otherwise the `jobNumber` and `password` are used in subsequent calls that manage the job, such as the following to retrieve real-time output from a NEOS solver while a job is running:

```
(msg, offset) =
   neos.getIntermediateResults(jobNumber, password, offset)
```

The `base64`-encoded `msg` contains the results of the job from character number `offset` up to the last received data. If the job is still running, then this function hangs until the next packet of output is sent or the job finishes. A non-blocking call is also provided and there are analogous functions for retrieving final output. An elementary Python script for a complete NEOS Server agent using these functions can be as simple as the program shown in Figure 1.

API calls are also available to retrieve information about the state of the NEOS Server, to obtain the list of solvers, and to kill a submission currently being executed. Calls of another kind register and maintain solvers on the Server.

Agents. Although XML document creation and NEOS Server communication can be carried out explicitly by a user, most of the time these activities are conducted by an intermediary program, called an *agent,* that makes the appropriate NEOS API calls.

As an example, consider what happens when a request is submitted to NEOS via the web form interface. Clicking the "Submit to NEOS" button initiates an agent (a Python CGI script) that makes a NEOS API call to get a list of the XML tags used by the selected solver, reads the data that the user has entered through text boxes, check boxes, and radio buttons of the web form, and combines this information to create an XML document. The agent then makes further NEOS API calls to submit the XML document and to check the Server intermittently for output and

```
#!/usr/bin/env python
import sys
import xmlrpclib

from config import Variables

if len(sys.argv) < 2 or len(sys.argv) > 3:
  sys.stderr.write("Usage: NeosClient <xmlfilename | help | queue>")
  sys.exit(1)

neos = xmlrpclib.ServerProxy("http://neos.mcs.anl.gov:3332")

msg = neos.ping()
if not msg.find("OK"):
  sys.stderr.write("Could not contact NEOS server.")
  sys.exit(1)

if sys.argv[1] == "help":
  sys.stdout.write("Help not yet available... ")

elif sys.argv[1] == "queue":
  msg = neos.printQueue()
  sys.stdout.write(msg)

else:
  xmlfile = open(sys.argv[1], "r")
  xmlstring = ""
  buffer = 1
  while buffer:
    buffer =  xmlfile.read()
    xmlstring += buffer
  xmlfile.close()

  (jobNumber, password) = neos.submitJob(xmlstring)
  sys.stdout.write("jobNumber = %d " % jobNumber)

  offset = 0
  status = neos.getJobStatus(jobNumber, password)
  while status == "Waiting" or status == "Running":
    (msg, offset) = neos.getIntermediateResults(jobNumber, password, offset)
    sys.stdout.write(msg.data)
    status = neos.getJobStatus(jobNumber, password)

  msg = neos.getFinalResults(jobNumber, password).data
  sys.stdout.write(msg)
```

Figure 1. *An elementary Python script for a complete NEOS Server agent.*

job status. When the job is finished, the agent directs the user's web browser to the results page. Figure 2 shows how different NEOS agents facilitate communications between local systems and the NEOS Server.

Kestrel agents for modeling systems. The mechanism for communication between a modeling system and NEOS is also implemented by creating an agent. In this case the agent is responsible for gathering a problem instance and related information from the modeling system, generating an XML document from this information, submitting the XML document to the NEOS Server, getting listings and results from the Server, and converting the results to a format that the modeling

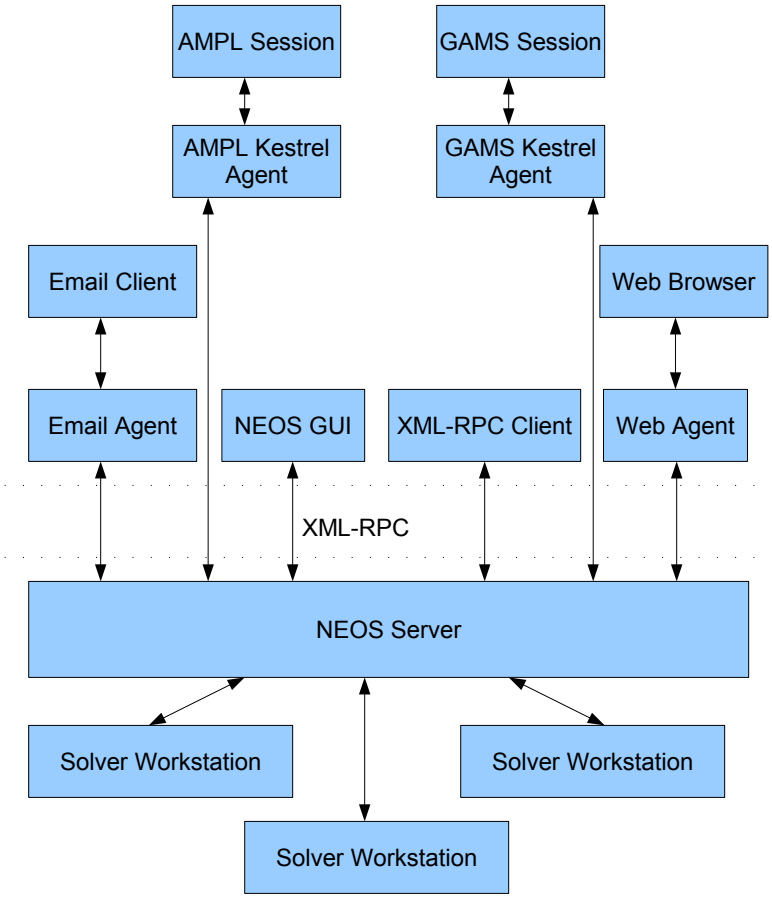

Figure 2. *Structure of the various NEOS interfaces, showing the connections between the NEOS Server and possible agents using the XML-RPC API.*

system can use.

Our implementations of such agents, initially for the AMPL and GAMS systems, are written in the Python programming language. Python has several features that make it ideal for this purpose:

- ◇ It is freely available and compatible with most computing platforms.
- ◇ It is distributed with an easy-to-use XML-RPC library.
- ◇ Third-party tools are available for converting Python programs to executable binaries for Microsoft Windows.

The Python code for our Kestrel agents for AMPL and GAMS can be downloaded from the NEOS web pages (see `neos.mcs.anl.gov/neos/downloads.html`), but a few key code fragments are presented here to give an idea of the issues involved in getting communication via Kestrel to work. Section 4 complements this discussion by describing how the Kestrel agents appear from the user's standpoint.

Although the AMPL and GAMS agents are separate programs, their code dealing with NEOS communications is similar. The following fragments are taken from the AMPL agent, but their GAMS equivalents are much the same.

The AMPL Kestrel agent is invoked when AMPL's `solve` command is given. First the agent encodes information about the problem instance, solver options, and related environment variables into an XML text document stored as a string in the Python variable `xml`. The agent then executes the following lines of code:

```
import xmlrpclib
self.neos = xmlrpclib.ServerProxy(
            "http://%s:%d" % (Variables.NEOS_HOST, Variables.NEOS_PORT))
```

These statements load the XML-RPC module `xmlrpclib` included with the Python distribution and create an object `self.neos` that will be used for communicating with the NEOS Server, which is listening at the URL given by `Variables.NEOS_HOST` and `Variables.NEOS_PORT` (actually `http://neos.mcs.anl.gov:3332`).

Communication with the Server can now begin. First the agent must check that the Server is running and accepting connections by using the API's `ping` call:

```
self.neos.ping()
```

Next the agent uses the `submitJob` call to send the XML document stored in the string variable `xml` to the NEOS Server:

```
(jobNumber, password) = self.neos.submitJob(xml)
```

When the Server receives this call, it performs the tasks necessary for initiating a job and returns a job number and password if the submission is successful, or a zero and an error message if an exception occured.

Finally, several calls are used to wait until the NEOS submission is finished, and to retrieve the results:

```
while status == "Running" or status == "Waiting":
   status = kestrel.neos.getJobStatus(jobNumber, password)
   time.sleep(5)
results = self.neos.getFinalResults(jobNumber, password)
if isinstance(results, xmlrpclib.Binary):
   results = results.data
```

Because the results for a Kestrel submission are returned as a `base64`-encoded string, the `data` data member of the return value is used to get the binary value represented by the string. This value is then written to a file from which the AMPL modeling environment extracts the solution.

4. Using the Kestrel Modeling Language Agents

The great majority of current submissions to the NEOS Server are written in the AMPL or GAMS modeling language (Figure 3). Yet as previously remarked, some of the advantages of a modeling language are lost when it serves merely as a NEOS input format. The NEOS Server proceeds in such a case as follows:

- ◇ The Server's input is passed to a remote workstation that has a modeling language *processor* as well as the requested *solver.*
- ◇ The modeling language processor interprets the input, converts it to a format that the solver recognizes, and invokes the solver.

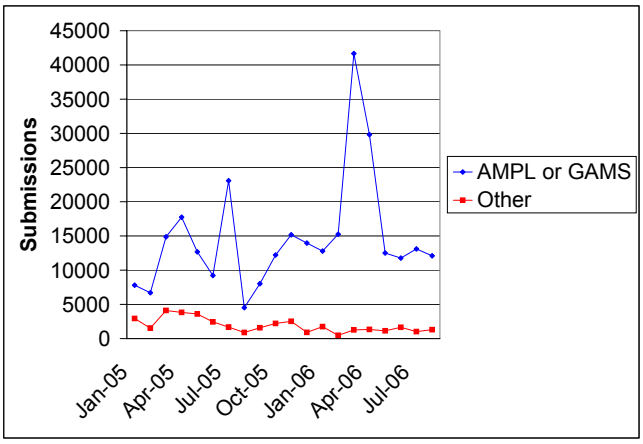

Figure 3. *Predominance of AMPL and GAMS in submissions to the NEOS Server.*

- ◇ When the solver run is complete, the processor retrieves results and optionally generates result tables specified by additional modeling language commands.

For security reasons, the modeling language processor runs in a mode that suppresses file writing and other local features. Thus only the solver's output and the processor's result tables are returned through the NEOS Server. Any interactivity, database connectivity, or other flexibility that the modeling system can provide when locally installed is lost in the Server environment.

Thus, one of the Kestrel interface's principal design requirements was to permit locally running GAMS and AMPL processes to call remote NEOS solvers in a way that preserves all of the familiarity and power of the GAMS and AMPL environments. This section first describes the use of the Kestrel/AMPL and Kestrel/GAMS agents for such a purpose, and then considers their advantages over other client-server arrangements for optimization.

Using the Kestrel/AMPL agent. AMPL specifies the solver name and options through a mechanism modeled on Unix and Windows environment variables and employs temporary files for the exchange of all other information with a solver. Thus, from the AMPL modeling system's standpoint, the invocation of a solver proceeds in three main steps:

- ◇ Write a representation of the current problem instance to a file, and set appropriate environment variables.
- ◇ Locate a specified *local solver,* invoke it as an independent program with appropriate arguments, and wait for it to write a solution file.
- ◇ Read the solution file and resume processing.

The Kestrel interface for AMPL substitutes the Kestrel/AMPL agent program for the local solver in the second step. The agent obtains a solution to the problem instance via the NEOS Server, rather than from some locally installed solver. But to the AMPL system, the agent is just another program that accepts problem instances and returns solutions. Thus, no special modification of AMPL is necessary to make it compatible with the Kestrel/AMPL agent.

As an example, the AMPL commands for solving the `steel.mod` sample problem (`www.ampl.com/BOOK/EXAMPLES2/steel.mod`) by using a locally installed copy of the LOQO solver [34] are as follows:

```
ampl: model steel.mod;
ampl: data steel.dat;
ampl: option solver loqo;
ampl: option loqo_options 'mindeg sigfig=8 outlev=1';
ampl: solve;
```

The corresponding commands to solve the same problem by using Kestrel/AMPL to run LOQO through the NEOS Server are much the same:

```
ampl: model steel.mod;
ampl: data steel.dat;
ampl: option solver kestrel;
ampl: option kestrel_options 'solver=loqo';
ampl: option loqo_options 'mindeg sigfig=8 outlev=1';
ampl: solve;
```

The `solver` option is changed from `loqo` to `kestrel`, so that the AMPL `solve` command now invokes the local Kestrel agent program. Also a command setting `kestrel_options` is added, to tell the Kestrel interface which remote NEOS solver is being requested. In all other respects, the AMPL session to this point is the same as before.

Once the problem instance has been successfully conveyed to the NEOS Server, the Kestrel agent displays the assigned job number and password:

```
Job 831764 sent to newton.mcs.anl.gov
password: ZQPNBVCc
```

Then it starts to echo any text that the solver subsequently writes to the standard output and error streams:

```
---------- Begin Solver Output -----------
Executing /home/neosotc/neos-5-solvers/kestrel-ampl/kestrelSolver.py
kestrel_options:solver=loqo

loqo_options:mindeg sigfig=8 outlev=1

LOQO 6.06: mindeg
sigfig=8
outlev=1

...

LOQO 6.06: optimal solution (11 QP iterations, 11 evaluations)
primal objective 191999.9989204328
  dual objective 191999.996903256
ampl:
```

When AMPL is being used interactively, text continues to appear a few lines at a time as the solver runs, with the frequency depending on how the solver manages its buffers. After the solver run is completed, all solution values are returned to the AMPL system and can be accessed through subsequent commands. The `display` command can be used to browse the solution, for example:

Figure 4. *The NEOS Server's web form for requesting results from a solver. In response the Server generates a page containing the current output or final job results, as appropriate.*

```
ampl: display Make;
Make [*] :=
bands  6000
coils  1400
;
```

The state of the AMPL system is in fact exactly the same at this point (except for the setting of the `solver` option) as it would be if the solver had been run locally.

Echoing the algorithm progress log interactively, as the solver generates it, is a substantial enhancement made possible by the decision to use XML-RPC as the communication protocol for NEOS 5. Earlier versions could only return the solver's output all at once, after the solver run was complete. To view current output, the job number and password could be entered into a web form (Figure 4) so that the results could be viewed through a browser, though they could still not be returned to the local AMPL environment. This facility remains available in NEOS 5, but is mainly of value for submissions through noninteractive interfaces such as email.

This enhancement extends to situations in which the local Kestrel or AMPL process is terminated and then restarted while the solver continues to run. Our design of Kestrel/AMPL for this situation is discussed in Section 6 in the context of the Kestrel modeling language agents' submission and retrieval features.

Using the Kestrel/GAMS agent. The GAMS modeling system communicates with solvers entirely via temporary files, so its communication with the Kestrel interface is also file-based. Prior to sending the GAMS problem file to the NEOS Server, the Kestrel/GAMS agent removes all references to the license and other system components, so that proper licenses and information for the remote solver can be inserted later. This practice helps preserve the integrity of the client's system by not sending license information over the Internet unnecessarily. The Kestrel/GAMS agent preprocessing also converts all absolute path names to relative ones.

To run the GAMSLIB `trnsport` model (`www.gams.com/modlib/libhtml/trnsport.htm`), as an example, the command `gams trnsport` is issued. The GAMS system then looks for additional commands within the file `trnsport.gms`. To solve the problem by using a local copy of the MINOS solver, the relevant commands are as follows:

```
model transport /all/;
option lp = minos;
solve transport using lp minimizing z;
```

To solve the same problem using MINOS remotely through the NEOS Server, we simply change the linear programming solver to `kestrel`:

```
model transport /all/;
transport.optfile = 1;
option lp = kestrel;
solve transport using lp minimizing z;
```

The added statement `transport.optfile = 1` specifies that an options file, named `kestrel.opt`, is provided. This options file conveys the remote solver name as well as any algorithmic directives for the remote solver. Thus for our MINOS example, `kestrel.opt` includes the line

```
kestrel_solver minos
```

and optionally any MINOS directives on additional lines.

After the problem instance has been submitted, the Kestrel/GAMS agent displays a confirmation with a job number and password, and the URL of a page where current or final results can be viewed. After the remote solver finishes its work, the solution is returned to GAMS, which may further process and report it just as if the problem had been solved locally.

Advantages of the Kestrel modeling language agents. Some work and possibly expense is involved in preparing to use the Kestrel interface. The prospective user who does not already have local access to AMPL or GAMS must acquire and install a copy—no special version is needed—either through purchase or by downloading a freely available version limited to 300 variables and constraints (and for GAMS also 2000 nonzeros, 1000 nonlinear nonzeros, and 50 integer variables).

Also Python, available without charge for all popular platforms, must be installed (see `www.python.org/download`), and a Python script implementing the AMPL or GAMS Kestrel agent must be downloaded (from `neos.mcs.anl.gov/neos/kestrel.html`). The Python scripts are only about 300–350 lines. We also offer the option of compiled Windows versions of the Python scripts, in the form of application and auxiliary files occupying 4–5 megabytes in total.

Once this initial investment is made, the local user gains access to all the NEOS solvers that have modeling language interfaces—18 at current writing—while retaining the benefits of the local features of the AMPL or GAMS modeling system. The interactive commands of AMPL remain available, for instance, to permit browsing through the solution and experimentation with changes to the model or data. Multiple files can also be read or written, optionally with connections to databases, spreadsheets, or the user's own programs.

In the early stages of developing an application, the Kestrel agents also offer the

advantage of allowing syntax and logic errors in the model and data to be caught locally. Certain infeasibilities can also be detected by AMPL's local presolve phase. Only valid problem instances reach the NEOS Server via the Kestrel agent, whereas other NEOS interfaces receive numerous AMPL and GAMS submissions that result in nothing more than syntax error reports.

Finally, the Kestrel interface can offer a considerable degree of security, because it sends NEOS only problem instances in low-level (usually binary) formats, and receives results only in similar formats. The other NEOS modes require models and data to be sent in text forms that people can read—indeed, these forms have been designed to promote readability—and are limited to producing results in readable listings as noted previously.

5. Representative Further Uses

Optimization applications often take input from numerous sources and require specific solver output for further postprocessing. Such requirements make it difficult, if not impossible, to use the classic NEOS submission mechanisms, but the Kestrel interface works well in these cases. As an example, this section describes an integration of Kestrel and other software to compute a so-called mountain pass solution to a semilinear partial differential equation.

The availability of the Kestrel interface also opens up possibilities for NEOS services that involve more than one resource. In particular, an optimization problem may be preprocessed in some manner prior to submitting it to the final solver for solution. The latter part of this chapter presents an example in which Kestrel preprocessing is used by a "GAMS/AMPL solver" on the NEOS Server.

Integrating with other software. Many applications need to interact with third-party software to generate problem data and to validate and visualize the results obtained from an optimization problem. This software cannot be provided to the NEOS Server, and is awkward to integrate with the inputs and outputs of the classical NEOS interfaces. Kestrel can provide an ideal mechanism for such applications.

As an example, we consider a case where Kestrel could serve to integrate NEOS with MATLAB for finding nontrivial solutions to a semilinear partial differential equation. We consider specifically the Lane-Emden-Fowler equation

$$-\Delta u = u^p,$$

with the Dirichlet boundary conditions $u(x) = 0$ for all $x \in \partial\Omega$, where u is the function that we are trying to find, Ω is the domain, $\partial\Omega$ is the boundary of the domain, and $p > 0$ is a constant exponent. Instead of solving this problem directly, we construct the variational function

$$F(u) = \int_\Omega \left(\|\nabla u(x)\|^2 - \frac{u(x)^{p+1}}{p+1} \right) dx$$

with the Dirichlet boundary conditions. All of this function's nontrivial critical points (where $u^* \neq 0$) correspond to solutions to the original partial differential equation.

Subject to reasonable assumptions, the Mountain Pass Theorem [28] states that between any two local minimizers of some function $f : \mathbb{R}^n \to \mathbb{R}$, there exists a continuous path connecting these minimizers whose maximum value is minimal over all such paths. The critical point where the maximum value is achieved along this optimal path is termed the *mountain pass.* A mountain pass is not a locally optimal point of f, but is rather a saddle point that gives the minimal change in function value needed to make a transition from one minimizer to another. In our example, *each mountain pass of F is a critical point of F,* so that solving the original partial differential equations corresponds to finding mountain passes of F.

We compute mountain passes by use of the *elastic string algorithm* [28]. First we approximate the variational function F by discretizing the domain Ω into a finite number of triangular elements and applying the trapezoidal rule to approximate the integral on each of these elements. The approximate variational function, $f(u)$, is then the summation over all triangular elements of the approximate integral over the elements. Using $f(u)$, the elastic string algorithm solves the following nonlinear optimization problem to find an approximation to a mountain pass:

$$\begin{array}{lll} \underset{u_1,\ldots,u_K,w}{\text{Minimize}} & w & \\ \text{Subject to} & w \geq f(u_k), & k = 1, \ldots, K \\ & \|u_k - u_{k+1}\|^2 \leq h_k^2, & k = 0, \ldots, K \\ & u_0 = u_a, & \\ & u_{K+1} = u_b, & \end{array}$$

Here K is the number of breakpoints in the piecewise-linear discretization of the path, u_k is a finite-dimensional approximation to u, $\|\cdot\|$ is the Euclidean norm, h_k is a given parameter constraining the distance between successive points, and u_a and u_b are two minimizers of F separated by a mountain range.

For this particular application, we fix the number of breakpoints to a relatively small number and solve the optimization problem to obtain an approximate mountain pass. Then we use an efficient Newton method to compute an accurate mountain pass. The key steps are as follows:

(1) Construct the finite-dimensional approximation to the variational problem by decomposing the given domain into elements, using a meshing package such as TRIANGLE [32].

(2) Calculate the starting and ending points u_a and u_b.

(3) Choose parameters K and h for the elastic string algorithm.

(4) Solve the nonlinear optimization problem.

(5) Compute an accurate solution in MATLAB [8, 30], and analyze and visualize the results.

Only steps (2)–(4) could be carried out remotely by the standard NEOS scheme of submitting modeling language declarations and commands. The decomposition in step (1) requires specialized local software and step (5) requires the execution of modeling language commands to produce a local file of descriptions that a MATLAB session can use to perform function, gradient, and Hessian evaluations.

Kestrel thus provides the interface of choice for this situation. Only step (4) is performed remotely on the NEOS Server by use of the Kestrel/AMPL agent as

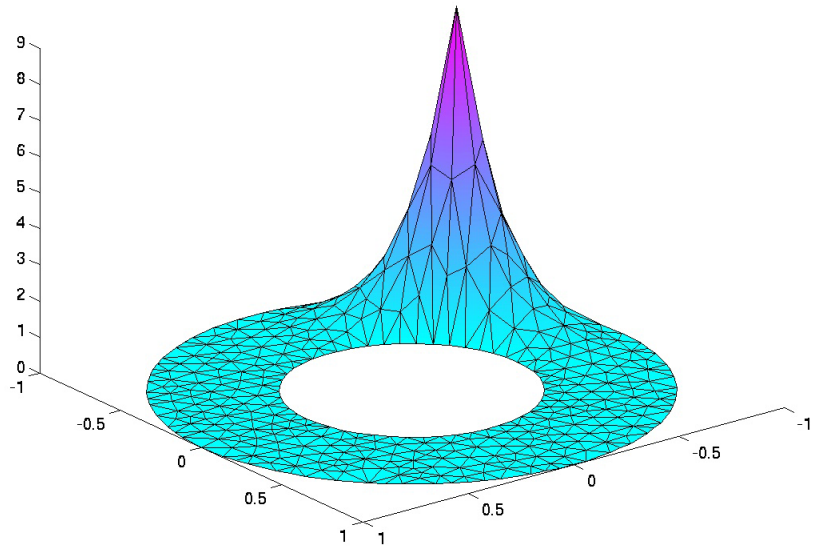

Figure 5. *Nontrivial solution to the Lane-Emden-Fowler equation with* $p = 3$.

previously described. The rest of the arrangement uses MATLAB and other locally executing programs. At the beginning of step (5) a final AMPL command writes a local file in AMPL's standard format for representation of nonlinear optimization problem instances, which is read by means of a MATLAB *mex* file as described in [21].

A complete AMPL model containing several sample domains, and the MATLAB scripts for the analysis, are available from `www.mcs.anl.gov/~tmunson/models/mountain.zip`. A nontrivial solution to the Lane-Emden-Fowler equation with $p = 3$ on an annulus domain generated by this code is depicted in Figure 5.

Preprocessor services on the NEOS Server. Preprocessing techniques are applied to an optimization problem instance to make it more amenable to solution, such as by reducing the number of variables and constraints [1, 6, 16, 31] or translating it to a different form. Thus, rather than submit a problem directly to a solver, a user may wish to submit it to some kind of preprocessor that is available through the NEOS Server. After simplifying or converting the problem in some way, the preprocessor can then use a Kestrel agent to submit the modified problem to a NEOS solver, without further intervention by the user. Results from the solver are returned via the Kestrel agent to the preprocessor, which can optionally undo some effects of the preprocessing before returning results to the user. The preprocessor and the solver need not reside on the same machine or have anything else in common other than their availability through the NEOS Server.

This arrangement is currently employed to provide a way of submitting GAMS models to Kestrel-enabled solvers requiring AMPL input. The preprocessor is the GAMS/CONVERT tool (`www.gamsworld.org/translate.htm`), which takes a GAMS model and data as input, and produces as output a *scalar* model—having no index sets—in any of several languages and formats. The NEOS Server uses this tool to offer a "GAMS/AMPL solver" that proceeds as follows:

(1) The user submits a GAMS problem file and an AMPL solver choice to the GAMS/AMPL solver.

(2) The Server directs the GAMS file and the solver choice to a remote workstation where GAMS, the GAMS/CONVERT preprocessor, and AMPL are available.

(3) The remote workstation initiates a GAMS process that translates the GAMS file and applies GAMS/CONVERT to produce an equivalent problem instance in scalar AMPL format.

(4) The GAMS process invokes AMPL to translate the scalar AMPL model to an internal AMPL representation, and then sends the internal AMPL representation and the solver choice back to the Server via a Kestrel/AMPL agent.

(5) The Server sends the internal representation of the scalar AMPL model to a second remote workstation where the chosen solver is available.

Once the solver finishes its work, these steps are reversed. The second remote workstation returns a solution file to the NEOS Server, which returns it to the first remote workstation, where GAMS/CONVERT translates the results to the form GAMS expects. The GAMS process then completes its work by sending the requested output back to the user via the Server.

The initial submission may be made through any of the NEOS interfaces. In particular, when the initial submission is itself made through the Kestrel/GAMS interface, then the single line like

```
kestrel_solver minos
```

in the GAMS `kestrel.opt` file described by Section 4 need only be replaced by the corresponding two lines

```
kestrel_solver gams_ampl
option solver loqo;
```

where LOQO is an example of a solver that supports AMPL but not GAMS input. Subsequent lines may contain other AMPL `option` statements. When the submission to the GAMS/AMPL solver is made in this way, the problem instance passes twice through the NEOS Server. Because translation of the GAMS model and data to a problem instance is done locally before the Kestrel submission, however, no translation is necessary in step (3).

6. Submission and Retrieval Features of the Kestrel Modeling Language Agents

We have assumed in previous examples that the local processes—AMPL or GAMS, and the corresponding Kestrel agent—remain active until results from the remote solver have been returned. This arrangement works well for a user whose number of submissions and solution time per submission are moderate. In other circumstances, however, the Kestrel facilities can be made considerably more useful and fault-tolerant by allowing job submission and retrieval to be requested separately. Should a failure occur on the local machine or in the NEOS Server while a problem is being solved on a remote workstation, the user can request results again at a later time after the failure has been corrected. The user may also deliberately

terminate the Kestrel agent and perform other modeling tasks before requesting the results of a solve that is expected to take a long time. The user's interim tasks may even include the submission of other problems through the Kestrel interface; by automating this possibility, we have been able to go beyond our original design goals by providing a rudimentary form of parallel processing.

In this section, we first describe two simple Kestrel extensions: one for retrieving results from submissions that have become disconnected, and the other for requesting cancelation of a submission. We then discuss the enhanced utilities for managing multiple submissions through Kestrel.

Retrieving disconnected submissions. Inevitably some Kestrel agent processes terminate prematurely. If the termination of a Kestrel agent occurs after the problem-submission call has completed, however, then further operations on the NEOS Server side are not affected. In particular, the NEOS Server still queues the problem and solves it on a remote workstation.

Since the web pages of intermediate results are generated by the NEOS Server independently of how the submission was made, termination of the Kestrel agent does not prevent such results from being viewed through a web interface like the one shown in Figure 4. Following completion of the remote solver's run, the complete result listing can be viewed through the same web interface for as long as the NEOS Server keeps it on file—currently a few days.

Termination of Kestrel/AMPL or Kestrel/GAMS also has no effect on the processes that eventually return the results from the NEOS Server. Thus the local modeling system that requested the results can still obtain them, provided that it can restart the Kestrel agent and can instruct it to retrieve a previous submission rather than begin a new one. The submission of interest can be identified by its job number and password, such as `831764` and `ZQPNBVCc` from the example in Section 4. This option proved easy to build into our modeling language agents.

With AMPL, the only change is to set `kestrel_options` to provide the job number and password rather than the solver name:

```
ampl: model steel.mod;
ampl: data steel.dat;
ampl: option solver kestrel;
ampl: option kestrel_options 'job=831764 password=ZQPNBVCc';
ampl: solve;
```

Kestrel then echoes all of the solver text from the beginning up to the current point for the requested job. If the solver has already finished, then all of its output is echoed at once. Either way, the solver's results are returned to the AMPL environment for viewing and processing using the usual AMPL commands. The ability to restart output from a solver that's still running is another enhancement made possible by the use of XML-RPC.

The corresponding extension for GAMS is equally straightforward. The file `kestrel.opt` is extended to give the job number and password,

```
kestrel_job 574537
kestrel_password TWHPGjdL
```

The command `gams trnsport` is then issued as before.

Terminating submissions. Occasionally a user wants to abandon a long-running submission because intermediate output shows little progress being made or because an error in the model or data has come to light. Merely terminating the current Kestrel agent process does not have this effect, as the preceding remarks have made clear.

Instead the Kestrel agent must be provided with a separate "kill" mode that makes a termination request rather than a result-retrieval request along with the job number and password. The request is communicated to the NEOS Server, which attempts to pass it to the workstation running the solver. (Certain combinations of workstation and solver are not configured to recognize termination requests.)

The mechanisms of the modeling languages are readily adapted to invoke the kill mode. For GAMS, we change the designation of the solver from `kestrel` to `kestrelkil` (with one final `l` due to a GAMS limitation of 10-characters per solver name). The job number and password are placed into the `kestrel.opt` file as before. Because the `kestrelkil` "solver" is invoked by a GAMS `solve` command, a model must be declared before this facility can be used in GAMS.

In the case of AMPL, `kestrel_options` is set as above, and a command script `kestrelkill` is invoked:

```
ampl: option kestrel_options 'job=831764 password=ZQPNBVCc';
ampl: commands kestrelkill;
```

The script resides in a file named `kestrelkill` in the current working directory. It has just one line,

```
shell 'kestrel kill';
```

that runs the quoted Kestrel agent command. We might have preferred to pass all of the information to the agent through `kestrel_options`, by recognizing a setting of, say, `'kill job=674537 password=TWHPGjdL'` followed by a `solve` command to invoke the Kestrel "solver" as before. AMPL's `solve` is not convenient for this purpose, however, because it refuses to invoke a solver until a model and data have been translated into a current problem instance. The `kestrelkill` script can be run even when no model and data have been read in the current session.

Managing submissions and retrievals. Most optimization modeling systems provide a way of defining *iterative schemes* that solve one or more models repeatedly. Common examples include sensitivity analysis, cross validation, decomposition, and row or column generation.

For these purposes the modeling language is extended to provide typical programming statements—such as assignments, tests, and loops—that use the same indexing and expressions as the language's model declarations. Programs, or *scripts*, using the new statements also have access to any of the commands already available for solving and manipulating problem instances. In particular, all of the commands we have shown for use of the Kestrel interface can be employed within a script, allowing scripts to do some processing on the local computer while sending optimization runs to NEOS solvers.

Many iterative schemes involve, in whole or part, the solution of successive collections of similar but independent *subproblems.* Since the NEOS Server typically has multiple workstations available to run the required solver, we might prefer to

send batches of subproblems to the NEOS Server, rather than submitting each subproblem and waiting for its solution before submitting the next. The effect is an elementary form of parallel processing that uses the NEOS Server's scheduler to manage the parallelism.

Because the Kestrel agent's problem-submission calls are separate from its result-retrieval calls, we can adapt the approach that Ferris and Munson [15] employed in their scheme for using modeling-language scripts to control Condor [27] workstation pools. Basically, a "solve" command can be replaced by separate "submit" and "retrieve" scripts or commands. Any number of submission requests are permitted, with retrievals taking place in the same order.

As an example, a conventional AMPL script for Benders decomposition of a two-stage stochastic location-transportation problem would contain an inner loop that solves one subproblem per pass:

```
for {s in SCEN} {
   let S := s;
   solve;
   let Exp_Ship_Cost := Exp_Ship_Cost + prob[S] * Scen_Ship_Cost;
   let {i in WHSE}
      supply_price[i,s,nCUT] := prob[S] * Supply[i].dual;
   let {j in STOR}
      demand_price[j,s,nCUT] := prob[S] * Demand[j].dual;
}
```

Because the subproblems are all independent linear programs, the subproblem processing may instead be broken into two loops, one that submits all the subproblems,

```
for {s in SCEN} {
   let S := s;
   commands kestrelsub;
}
```

followed by one that retrieves all the corresponding results:

```
for {s in SCEN} {
   let S := s;
   commands kestrelret;
   let Exp_Ship_Cost := Exp_Ship_Cost + prob[S] * Scen_Ship_Cost;
   let {i in ORIG}
      supply_price[i,s,nCUT] := prob[S] * Supply[i].dual;
   let {j in DEST}
      demand_price[j,s,nCUT] := prob[S] * Demand[j].dual;
}
```

By indexing over the same set in both loops, we ensure that retrieval of problem results occurs in the same order as problem submission.

The AMPL Kestrel agent implements this behavior by use of a single temporary job file. The `kestrelsub` script is just three lines:

```
option ampl_id (_pid);
write bkestproblem;
shell 'kestrel submit kestproblem';
```

The first line serves only to insure that all invocations of the agent refer to the

same temporary file, whose name is constructed from the parameter `_pid` that AMPL predefines to equal the process ID of the current AMPL session. The `write` command generates a binary problem instance in a file `kestproblem.nl`—just as a `solve` command would do, but without automatically invoking any solver. Instead the `shell` command invokes a Kestrel agent process with the argument `submit` and the problem file name. In this mode Kestrel/AMPL reads the file, submits it to the Server, appends the resulting job number and password to the job file, and then terminates without waiting for a result.

The complementary `kestrelret` script performs analogous actions in reverse order. It too is just three lines:

```
option ampl_id (_pid);
shell 'kestrel retrieve kestresult';
solution kestresult.sol;
```

The first line again serves to insure that all invocations of the agent refer to the same temporary file. Then `shell` invokes a Kestrel agent process but passes it the argument `retrieve` and a result file name `kestresult`. The agent process asks the NEOS Server for results from the first job listed in the job file, then waits. As soon as a response is received, the agent saves the results file as `kestresult.sol`, removes the first entry from the job file, and terminates. The final statement of the script runs the AMPL command `solution` to read the contents of the results file back into the AMPL system. (One `kestproblem.nl` and one `kestresult.sol` file remain in the current working directory at the end of the AMPL session and must be deleted by additional `shell` commands at the end of the iterative scheme.)

The same effects are achieved in the GAMS environment by the creation of two new "solvers"—`kestrelsub` and `kestrelret`—that implement the submission and retrieval parts of the Kestrel agent. For example, the following commands implement a sensitivity analysis scheme for the `trnsport` example:

```
SET iter /1*5/;
PARAMETER optval(iter);
PARAMETER avail(iter);

avail(iter) = a('seattle') + 10*ord(iter);

LOOP (iter,
   a('seattle') = avail(iter);
   option lp = kestrelsub;
   solve transport using lp minimizing z;
);

LOOP (iter,
   a('seattle') = avail(iter);
   option lp = kestrelret;
   solve transport using lp minimizing z;
   optval(iter) = z.l;
);
```

Again `kestrelsub` and `kestrelret` write job information to a temporary job file, in this case stored in the GAMS scratch directory and removed once processing completes. We break with GAMS convention by also using `kestrel.opt` as the options file for both, rather than looking for separate files `kestrelsub.opt` and `kestrelret.opt`.

The usefulness of this facility depends upon the relative processing costs of the NEOS Server and of the solver. The communication and scheduling costs would likely dominate the solution time for the Benders and `trnsport` examples if the linear programs generated were easily solved. More complex applications can be identified, however, in which it is advantageous to exploit this kind of parallelism. The 10-fold cross validation scheme from [15] is one example, because it requires solutions to many independent integer programming problems whose cost per solve has been observed to greatly exceed total costs for communication and scheduling.

The ability to use this facility to solve optimization subproblems in parallel is also necessarily limited by the resources available to the NEOS Server. When the number of jobs submitted for a solver exceeds the number that available workstations can handle, the Server queues additional jobs and starts them as soon as workstations become free. However, the Server imposes an upper bound on the number of submissions that may be queued for any one solver. Submissions to a solver that already has a full submission queue are then rejected. Schemes such as Benders decomposition might require subproblems to be aggregated so that they do not overwhelm the Server's queues.

7. Further Directions

Despite its overall complexity, the NEOS Server can provide an API that is both straightforward for programmers and powerful enough to support the varied needs of modelers. Building on the version 5 API, the Kestrel interface makes NEOS solvers available to disparate modeling languages and systems through only minor changes to their procedures for local solvers.

The Kestrel interface has also proved to have related uses that we did not anticipate. The GAMS/AMPL conversion tool described in Section 5 became possible only upon the realization that the NEOS Server could invoke itself via a "solver" that acted as a preprocessor. The kind of parallelism described in Section 6 takes advantage of features originally conceived to provide fault tolerance in the event of a broken connection.

While we have concentrated on general-purpose environments based on modeling languages, the same approach can also be used to make NEOS solvers available to systems or front-ends that are specialized for particular applications. We expect this to open up further unanticipated uses for the NEOS API and Kestrel interface.

Acknowledgment

"Kestrel" pays homage to Condor [27], a system for utilizing idle workstations, and to the work in [15] that provides a simple form of parallelism by way of a mechanism for submitting optimization problems to a Condor pool from a modeling environment.

References

[1] E. Andersen and K. Andersen, Presolving in linear programming. *Mathematical Programming*, **71** (1995) 221–245.

[2] J.R. Birge, M.A.H. Dempster, H.I. Gassmann, E.A. Gunn, A.J. King, and S.W. Wallace, A Standard Input Format for Multiperiod Stochastic Programs. *Mathematical*

Programming Society Committee on Algorithms Newsletter **17** (1987) 1–20; also at `www.mgmt.dal.ca/sba/profs/hgassmann/smps.html`.

[3] C. Bischof, A. Carle, P. Khademi, and A. Mauer, ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs. *IEEE Computational Science & Engineering* **3** (1996) 18–32.

[4] C. Bischof, L. Roh, and A. Mauer, ADIC – An Extensible Automatic Differentiation Tool for ANSI-C. *Software: Practice and Experience* **27** (1997) 1427–1456.

[5] J.J. Bisschop and A. Meeraus, On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study* **20** (1982) 1–29.

[6] A.L. Brearley, G. Mitra and H.P. Williams, Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method. *Mathematical Programming* **8** (1975) 54–83.

[7] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide, Release 2.25.* Scientific Press/Duxbury Press (1992).

[8] S.J. Chapman, *Matlab Programming for Engineers* (2nd edition). Brooks/Cole, Pacific Grove, CA (2002).

[9] A.R. Conn, N.I.M. Gould and Ph.L. Toint, *LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization.* Springer Verlag (1992).

[10] J. Czyzyk, M.P. Mesnier and J.J. Moré, The NEOS Server. *IEEE Journal on Computational Science and Engineering* **5** (1998) 68–75.

[11] J. Czyzyk, J.H. Owen and S.J. Wright, Optimization on the Internet. *OR/MS Today* **24,** 5 (October 1997) 48–51; also at `www.mcs.anl.gov/otc/Guide/TechReports/otc97-04/orms.html`.

[12] E.D. Dolan, NEOS Server 4.0 Administrative Guide. Technical Memorandum ANL/MCS-TM-250, Mathematics and Computer Science Division, Argonne National Laboratory (2001); at `www-neos.mcs.anl.gov/neos/ftp/admin.ps.gz`.

[13] E.D. Dolan, R. Fourer, J.J. Moré and T.S. Munson, The NEOS Server for Optimization: Version 4 and Beyond. Preprint ANL/MCS-P947-0202, Mathematics and Computer Science Division, Argonne National Laboratory (2002); at `www-neos.mcs.anl.gov/neos/ftp/v4.pdf`.

[14] E.D. Dolan, R. Fourer, J.J. Moré and T.S. Munson, Optimization on the NEOS Server. *SIAM News* **35,** 6 (July/August 2002) 4, 8–9.

[15] M.C. Ferris and T.S. Munson, Modeling Languages and Condor: Metacomputing for Optimization. *Mathematical Programming* **88** (2000) 487–505.

[16] M.C. Ferris and T.S. Munson, Preprocessing Complementarity Problems. In *Complementarity: Applications, Algorithms and Extensions,* M.C. Ferris, O. Mangasarian, and J.S. Pang, eds., Kluwer Academic Publishers, Dordrecht, The Netherlands (2001) 143–164.

[17] R. Fourer, D.M. Gay and B.W. Kernighan, A Modeling Language for Mathematical Programming. *Management Science* **36** (1990) 519–554.

[18] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming* (2nd edition). Duxbury Press, Pacific Grove, CA (2003).

[19] R. Fourer and J.-P. Goux, Optimization as an Internet Resource. *Interfaces* **31,** 2 (March/April 2001) 130–150.

[20] K. Fujisawa, M. Kojima, and K. Nakata, SDPA (Semidefinite Programming Algorithm) User's Manual. Technical Report B-308, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology (1998).

[21] D.M. Gay, Hooking Your Solver to AMPL. Technical Report 97-4-06, Computing Sciences Research Center, Bell Laboratories (1997); at `www.ampl.com/REFS/hooking2.pdf`.

[22] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel and A. Walther, ADOL–C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Transactions on Mathematical Software* **22** (1996) 131–167.

[23] W. Gropp and J.J. Moré, Optimization Environments and the NEOS Server. In *Approximation Theory and Optimization,* M.D. Buhmann and A. Iserles, eds., Cambridge University Press (1997) 167–182.

[24] P.D. Hovland and B. Norris, Users' Guide to ADIC 1.1. Technical Memorandum ANL/MCS-TM-225, Argonne National Laboratory (2001).

[25] C.A.C. Kuip, Algebraic Languages for Mathematical Programming. *European Journal of Operational Research* **67** (1993) 25–51.

[26] J. Linderoth and S. Wright, Decomposition Algorithms for Stochastic Programming on a Computational Grid. *Computational Optimization and Applications* **24** (2003) 207–250.

[27] M.J. Litzkow, M. Livny and M.W. Mutka, Condor: A Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing Systems* (1988) 104–111.

[28] J.J. Moré and T.S. Munson, Computing Mountain Passes and Transition States. *Mathematical Programming* **100** (2004) 151–182.

[29] B.A. Murtagh, *Advanced Linear Programming: Computation and Practice*. McGraw-Hill International Book Company (1981).

[30] W.J. Palm III, *Introduction to Matlab 6 for Engineers*. McGraw-Hill, New York (2001).

[31] M. Savelsbergh, Preprocessing and Probing Techniques for Mixed Integer Programming problems. *ORSA Journal on Computing* **6** (1994) 445–454.

[32] J. Shewchuk, Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Proceedings of the First Workshop on Applied Computational Geometry,* Association for Computing Machinery (1996) 124–133.

[33] A. Skonnard and M. Gudgin, *Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More*. Addison-Wesley Professional (2001).

[34] R.J. Vanderbei and D.F. Shanno, An Interior Point Algorithm for Nonconvex Nonlinear Programming. *Computational Optimization and Applications* **13** (1999) 231–252.

[35] D. Winer, XML-RPC website, `www.xmlrpc.com` (2003).