Implementing an XML−RPC client in Mathematica

Dario Malchiodi

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
http://homes.dsi.unimi.it/~malchiod
mailto:malchiodi@dsi.unimi.it

XML−RPC is a protocol used to remotely execute a program independently of the particular hardware and operating system used on both ends of the communication channel, in that all the conveyed information consists in text containing XML encodings coupled with HTTP headers. A sagacious mix of *J/Link*, XML, and RegularExpression technologies available within *Mathematica* allows to easily implement a client exploiting this protocol.

■ 1. Introduction

The possibility of executing code remotely, i.e. sending the arguments from a given computer to another one actually performing the execution and sending back the return value, allows for a correct use of high−performance computing facilities, such as clusters [1] and grids [4]. This kind of interaction, initially introduced by Sun Microsystems [12], is known as Remote Procedure Call (RPC for short) and represents an implementation of the client−server model for distributed programming, which has been implemented in various forms (see for instance the SOAP [11] protocol).

In order to insure that the protocol is independent of the used computing environment, the client encodes the procedure's name and arguments using an *interface description language*, and this encoding is sent to the server through a given *network protocol*. The same protocol and language are then used to send back the procedure's return value.

■ 2. The XML−RPC protocol

XML−RPC is an implementation of RPC [12] encoding the interface description language through XML (eXtensible Markup Language) [14] and using HTTP (HyperText Transfer Protocol) [3] as network protocol. The basic idea is to handle RPCs using POST requests whose content is a XML encoding of the method name and of the related arguments.

The typical call is encoded through the following request, to be sent to an existing server:

```
POST /PRC2 HTTP/1.0
User−Agent: agent
Host: host
Content−Type: text/xml
```

```
Content-Length: length

<?xml version="1.0"?>
<methodCall>
  <methodName>method name</methodName>
  <params>
    <param>
      <value><type>value</type></value>
    </param>
    ...
  </params>
</methodCall>
```

where bold text denotes the following values to be adapted to each particular call:

- **agent** and **host** are strings qualifying respectively: i) the environment from within the call is requested (typically the used software name, release, and operating system), and ii) the corresponding host location (in terms of a symbolic or numeric IP);
- **length** counts the number of bytes of the call *payload* (the following XML structure);
- **method name** is the name of the procedure to be executed;
- **value** represents the value of the method's first argument. It is enclosed between nested `param` and `value` tags, containing a further tag whose name (denoted **type** in the above example) encodes the argument type. Supported types, both at the simple and structured level, are listed in Table 0.

  Each argument is encoded in a `param` tag, and their ordering must match the one specified in the called method's signature.

`int`, `i4`	signed integer (four bytes)
`boolean`	boolean (0 = false, 1 = true)
`string`	string
`double`	floating point number
`dateTime.iso8601`	date / time (in ISO 8601 format)
`base64`	base64-encoded binary
`array`	array of data
`struct`	XML structure

Supported data types in XML−RPC.

Consider for instance a `getSales` method, with intuitive meaning and signature `int getSales(String region, int year)`; the POST request corresponding to a call to this method is the following (using dummy values for the `User-Agent` and `Host` entries):

```
POST /PRC2 HTTP/1.0
User-Agent: XML-RPC client (OS)
Host: 10.0.0.1
Content-Type: text/xml
Content-Length: 240

<?xml version="1.0"?>
<methodCall>
```

```
  <methodName>getSales</methodName>
  <params>
    <param>
      <value><string>Europe</string></value>
    </param>
    <param>
      <value><int>2002</int></value>
    </param>
  </params>
</methodCall>
```

Once a request is sent, the server answers sending an HTTP document whose payload contains either the return value for the method or an error message. In the first case, the server returns

```
HTTP/1.0 200 OK
Server: server
Date: date
Content−Type: text/xml
Content−length: length

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><type>value</type></value>
    </param>
  </params>
</methodResponse>
```

where the entries typeset in bold have either an obvious meaning or are encoded as in the previous `<methodCall>` structure. Note that in this case the payload contains exactly one `param` tag, for a method has only one return value; Multiple return values can be dealt with using the `array` and `struct` tags [14].

For instance, a successful call to the `getSales` method introduced before could give rise to the following answer:

```
HTTP/1.0 2000 OK
Server: XML−RPC server (OS)
Date: Tue, 10 Jan 2006 22:30:48 GMT
Content−Type: text/xml
Content−length: 142

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><int>2564</int></value>
    </param>
  </params>
</methodResponse>
```

Unsuccesfull calls give rise to a different `methodResponse` tag, enclosing a `fault` tag instead of the `params` one. The former in turn contains a `struct` tag encoding an integer code and a textual description for the error.

■ 3. The *Mathematica* implementation

Implementing an XML–RPC client in *Mathematica* requests to set up a network channel as communication medium, generating and sending through the latter the header and payload for a given RPC, reading the corresponding answer, validating it against possible errors and finally converting the result into *Mathematica* expressions.

This implementation is available for download in form of a package [8], which can be used to extend *Mathematica*'s abilities of remotely accessing computational facilities, in that the available Web services package does not manage the XML–RPC protocol.

■ 4. Setting up the communication channel

The first step consists in launching the Java runtime and storing in two variables the address and port of the server. Throughout this paper we will quote as an application example the NEOS server for optimization [2], a distributed, multi–method optimization system with multiple interfaces, including a set of APIs accessible through XML–RPC.

```
<< JLink‘;
InstallJava[];
$xmlRpcServer = "neos.mcs.anl.gov"; $xmlRpcPort = 3332;
```

In order to provide a medium through which vehiculate XML–RPC requests and answers, socket–based communication needs to be set up. This can be done accessing the `java.net` and `java.io` packages through *J/Link:*

```
cli = JavaNew["java.net.Socket", $xmlRpcServer, $xmlRpcPort];
out =
  JavaNew["java.io.DataOutputStream", cli@getOutputStream[]];
in = JavaNew["java.io.DataInputStream", cli@getInputStream[]];
```

in this way, the `in` and `out` variables will point to an input and an output stream which will be used for sending XML–RPC requests and receiving the corresponding answers, respectively.

■ 5. Formatting requests

Requests to an XML–RPC server follow the format explained in Sec. 2, i.e. an HTTP header followed by an XML structure. Let us initially focus on the latter. Using the SymbolicXML format available within *Mathematica* it is possible to create its skeleton to be filled afterwards:

```
xmlTempl = ImportString["<?xml version=\"1.0\"?><methodCall><
     methodName/><params/></methodCall>", "XML"];
```

The simplest call is that to a method requesting no arguments. In this case the sole part of the skeleton to be filled is the empty `methodName` tag, then SymbolicXML can be translated into a string through the `ExportString` function. For instance, the following cell stores in the `payload1` variable the XML encoding of a call to the method `version`—used for testing purposes—which requests no arguments and returns a string describing the server version number:

```
methodName = "version";
xmlObj = xmlTempl /. XMLElement["methodName", {}, {}] →
    XMLElement["methodName", {}, {methodName}];
payload1 = ExportString[xmlObj, "XML"]
```

```
<?xml version='1.0'?>
<methodCall>
 <methodName>version</methodName>
 <params/>
</methodCall>
```

When a method requests one or more arguments, also the `params` tag need to be filled with a `param` tag for each argument: in view of an implementation independent of a particular method call, this can be accomplished storing the arguments in a list (where each argument is in turn a list containing its type and value), over which a suitable function is mapped. For instance, the following cell stores in the `payload2` variable the XML encoding of a call to the method `listSolversInCategory`, which returns a list of all the solvers pertaining to a given category, specified as argument (this example uses the category "nco", standing for "non−constrained optimization"):

```
methodName = "listSolversInCategory";
params = {{"string", "nco"}};
xmlObj = xmlTempl /. XMLElement["methodName", {}, {}] →
    XMLElement["methodName", {}, {methodName}];
xmlObj = xmlObj /. XMLElement["params", {}, {}] →
    XMLElement["params", {},
     Map[XMLElement["param", {}, {XMLElement["value", {},
          {XMLElement[#[[1]], {}, {#[[2]]}]}]}] &, params]];
payload2 = ExportString[xmlObj, "XML"]
```

```
<?xml version='1.0'?>
<methodCall>
 <methodName>listSolversInCategory</methodName>
 <params>
  <param>
   <value>
    <string>nco</string>
   </value>
  </param>
 </params>
</methodCall>
```

Once the payload has been built, the corresponding header is easily obtained after joining a set of strings—one per header entry—as follows:

- the `POST` and `Content−type` entries are rendered as constant strings;
- the `User−Agent` value specifies the *Mathematica* version and operating system, obtained respectively from the `$VersionNumber` and `$System` variables;
- the `Host` value consists in the joined values of the `$MachineName` and `$MachineDomain` variables;
- the `Content−Length` value is obtained as the length of the string containing the payload to be sent.

For instance, a header for the request contained in `payload1` defined above is built as follows:

```
head1 = "POST /RPC2 HTTP/1.0\n";
head1 = head1 <> "User-Agent: Mathematica/";
head1 = head1 <> ToString[$VersionNumber];
head1 = head1 <> " (" <> $System <> ")\n";
head1 = head1 <> "Host: " <> $MachineName;
head1 = head1 <> "." <> $MachineDomain <> "\n";
head1 = head1 <> "Content-Type: text/xml\n";
head1 = head1 <> "Content-Length: ";
head1 = head1 <> ToString[StringLength[payload1]] <> "\n";
head1 = head1 <> "\n";
```

so that the whole content to be sent to the server is

```
Print[head1 <> payload1];
```

```
POST /RPC2 HTTP/1.0
User-Agent: Mathematica/5.1 (Mac OS X)
Host: smirnov.
Content-Type: text/xml
Content-Length: 93

<?xml version='1.0'?>
<methodCall>
 <methodName>version</methodName>
 <params/>
</methodCall>
```

The same procedure applies to other headers, for instance the one corresponding to the variable `payload2`. The only difference possibly stands in the `Content-Length` entry.

■ 6. Sending requests and receiving answers

Sending a request is a matter of writing the previously obtained string on the socket output stream, namely calling the `writeBytes` method of the `DataOutputStream` class. This involves again the use of *J/Link*:

```
out@writeBytes[head1 <> payload1];
```

A dual action is requested to retrieve the return value of the called method, which will involve reading from the input stream contained in the `in` variable. Unlike the send process, in this case the amount of data to be read is unknown, thus the stream will be read one line at a time, until a `Null` value is returned, denoting the end of input:

```
answer = ""; fetch = in@readLine[];
While[StringQ[fetch],
  answer = answer <> fetch <> "\n";
  fetch = in@readLine[];];
```

As no more interaction with the server is required, the socket and the streams can be safely closed:

```
out@close[]; in@close[]; cli@close[];
```

Now, if the method has been correctly called, the `answer` variable contains the value returned by the called method. However, this value is encoded as explained in Sec. 2:

```
Print[answer];
```

```
HTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.4.1
Date: Sat, 14 Jan 2006 16:29:14 GMT
Content-type: text/xml
Content-length: 140

<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><string>neos version 5</string></value>
</param>
</params>
</methodResponse>
```

The first action to be performed on this return value consists in extracting the payload: this can be simply done after ignoring all the string contents until a couple of newlines is found:

```
retVal = StringReplace[answer,
  RegularExpression["(.|\n)*?\n\n((.|\n)*)"] -> "$2"]
```

```
<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><string>neos version 5</string></value>
</param>
</params>
</methodResponse>
```

The same execution on the `listSolversInCategory` (i.e. processing the value returned after having sent `head2<>payload2` on the communication channel) would bring to the following result:

```
<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><array><data>
<value><string>CONOPT:GAMS</string></value>
<value><string>filter:AMPL</string></value>
<value><string>Ipopt:AMPL</string></value>
<value><string>KNITRO:AMPL</string></value>
<value><string>KNITRO:GAMS</string></value>
<value><string>LANCELOT:AMPL</string></value>
<value><string>LANCELOT:SIF</string></value>
<value><string>LOQO:AMPL</string></value>
<value><string>MINOS:AMPL</string></value>
<value><string>MINOS:GAMS</string></value>
<value><string>MOSEK:AMPL</string></value>
<value><string>MOSEK:GAMS</string></value>
<value><string>PATHNLP:GAMS</string></value>
<value><string>PENNON:AMPL</string></value>
<value><string>SNOPT:AMPL</string></value>
<value><string>SNOPT:FORTRAN</string></value>
<value><string>SNOPT:GAMS</string></value>
</data></array></value>
</param>
</params>
</methodResponse>
```

Erroneus remote calls (such as one to a nonexistent method) are detected exploiting the different contents of the `methodResponse` tag. Namely, if the returned payload matches with the regular expression

```
___ ~~ "<params>" ~~ ___ ~~ "</params>" ~~ ___
```

the RPC has been correctly executed. Otherwise, an error occurred.

■ 7. Decoding output

The first action to be done with a string description obtained in Sec. 6 which wasn't filtered against errors is to extract the meaningful information contained in the `param` tag. This is accomplished, as in previous sections, using a suitable string matching procedure extracting the type and value—ever described using strings—of the corresponding return value and inserting them in a list.

```
StringReplace[retVal,
  ___ ~~ "<param>" ~~ Whitespace ... ~~ "<value>" ~~
       Whitespace ... ~~ "<" ~~ v : Except[">"] .. ~~
          ">" ~~ j___ ~~ "</" ~~ Except[">"] .. ~~ ">" ~~
               Whitespace ... ~~ "</value>" ~~ Whitespace ... ~~
                  "</param>" ~~ ___ → {v, j}][[1]]

{string, neos version 5}
```

On this list we will compute a function decoding the values into *Mathematica* expressions. This function has to handle the different types supported by the XML–RPC protocol. The implementation is straightforward for the simpler types: integer and floating–point values are esaily dealt with using the `ToExpression` function, boolean values are examined exaustively and strings are obviously left untouched. Analogously, the XML tree described by the `struct` data type can be automatically converted into an `XMLObject` using the `ImportString` function.

clear text	*A Mathematica* XML – RPC package
encoding	`QSBNYXRoZW1hdGljYSBYTUwtUlBDIHBhY2thZ2U=`

A string of text and the corresponding base64 encoding.

Base64 encoded values need a slightly more complex processing: this 64–bit encoding [6]—typically used in order to encode binary attachments to e–mails—maps a group of three consecutive bytes with four ASCII printable characters, as exemplified in Table 0. Thus the corresponding decoding consists of:

- a function implementing the decode table, mapping a single printable character from the encoded text into a byte;
- a function mapping a set of four consecutive characters in output of the function in previous point into the corresponding set of three bytes in the decoded text;
- a function partitioning the encoded text into successive groups of four characters and applying to the latter the above function (possibly discarding trailing '=' characters, used during the encoding phase to pad original sequences not amounting to a multiple of three bytes).

A similar procedure can be used in order to encode clear text using the same protocol [7].

Finally, the ISO8601 standard [5] for time and date can be easily processed: as the required format for a time/date is `yyyy-mm-ddThh:mm:ss`, we can overload

`ToDate` in order to accept as argument a string, whose contents will be split into a list (using the characters `-`, `:`, and `T` as separators) and then converted into numeric values.

Putting together those observations we obtain the `xmlRpcDecode` function, to be applied to the list obtained at the beggining of this section:

```
xmlRpcDecode[{type_, value_}] := Block[{},
  Switch[type,
   "i4" | "int" | "double", ToExpression[value],
   "boolean", If[value == "0", False, True],
   "dateTime.iso8601", ToDate[value],
   "base64", base64Decode[value],
   "array", arrayDecode[value],
   "struct",
   ImportString["<struct>" ~~ value ~~ "</struct>", "XML"],
   _, value]]
```

This function handles arrays through the following `arrayDecode` function, removing the leading and trailing `data` tags, associating to each `value` tag the corresponding *{type, value}* couple, and building a list containing the so obtained couples. Finally, `xmlRpcDecode` is recursively mapped on this list:

```
arrayDecode[value_] := StringCases[
  StringReplace[value, "<data>" ~~ j___ ~~ "</data>" → j],
  ShortestMatch[Whitespace ... ~~ "<value>" ~~ Whitespace ... ~~
       "<" ~~ v : (Except[">"] ..) ~~ ">" ~~ j___ ~~
           "</" ~~ Except[">"] .. ~~ ">" ~~ Whitespace ... ~~
               "</value>"] → xmlRpcDecode[{v, j}]]
```

■ 8. Gluing the components

Putting together the code shown so far, we obtain a set of functions enabling us to execute RPCs through one single function call:

- `xmlRpcInit`, to be called at the beginning of an XML−RPC session in order to fix address and port of the server;
- `xmlRpcPayload`, returning a string containing the payload for an XML−RPC request;
- `xmlRpcHeader`, returning the header corresponding to a given payload;
- `xmlRpcDecode`, translating the XML values returned from a remote call into valid *Mathematica* expressions;
- `xmlRpcGetAnswer`, extracting and converting into a *Mathematica* expression the meaninful information containted in the payload returned from an XML−RPC when no errors arise;
- `xmlRpc`, built over the above functions and executing an XML−RPC whose return value is automatically converted into a *Mathematica* expression.

Among all these functions, in order to access a given server only `xmlRpcInit` and `xmlRpc` will be directly called: the former once a new server needs to be addressed, and the latter for each call to this server.

The above functions constitute the xmlRpc package [8], available for download at the web page http://homes.dsi.unimi.it/~malchiod/software/xmlRpc.

■ 9. Examples

The neosAPI package

The described XML–RPC implementation allows to access within *Mathematica* to the procedures available on a given server. For instance, the `neosAPI` package [9] implements in this way a subset of procedures related to the above mentioned NEOS server for optimization. Consider for instance the constrained optimization problem:

$$\min(x^2 + y^2) \text{ such that } x + y = 3$$

The following cell builds a string containing the description of this problem in the AMPL modeling language [10]:

```
<< neosAPI`
r = "&lt;document&gt;\n";
r = r <> "&lt;category&gt;nco&lt;/category&gt;\n";
r = r <> "&lt;solver&gt;SNOPT&lt;/solver&gt;\n";
r = r <> "&lt;inputMethod&gt;AMPL&lt;/inputMethod&gt;\n";
r = r <> "&lt;model&gt;&lt;![CDATA[\n\n";
r = r <> "var x;\nvar y;\n";
r = r <> "minimize obj:x*x+y*y;\n";
r = r <> "subject to constr:x+y=3;\n";
r = r <> "]]&gt;&lt;/model&gt;\n\n";
r = r <> "&lt;data&gt;&lt;![CDATA[\n";
r = r <> "data;\n";
r = r <> "]]&gt;&lt;/data&gt;\n\n";
r = r <> "&lt;commands&gt;&lt;![CDATA[\n";
r = r <> "solve;\n";
r = r <> "]]&gt;&lt;/commands&gt;\n\n";
r = r <> "&lt;/document&gt;\n";
```

The use of HTML entities is a requirement of XML–RPC specifications [14], though a simple transformation can display the description in a more readable form:

```
StringReplace[r, {"&lt;" -> "<", "&gt;" -> ">"}]
```

```
<document>
<category>nco</category>
<solver>SNOPT</solver>
<inputMethod>AMPL</inputMethod>
<model><![CDATA[

var x;
var y;
minimize obj:x*x+y*y;
subject to constr:x+y=3;
]]></model>

<data><![CDATA[
data;
]]></data>

<commands><![CDATA[
solve;
]]></commands>

</document>
```

The problem is solved in three steps:

- the function `neosSubmitJob` sends the AMPL description to the NEOS server, which in turn places the corresponding job in a queue, returning the job's number and password:

```
{jobNum, jobPwd} = neosSubmitJob[r]
```

```
{730876, HklLMQPD}
```

- the function `neosGetJobStatus` allows for verifying whether or not the above job has been executed:

```
neosGetJobStatus[jobNum, jobPwd]

Done
```

- the function `neosGetFinalResults` retrieves the job output:

```
neosGetFinalResults[jobNum, jobPwd]

Job 730876 sent to schwinn.mcs.anl.gov
password: HklLMQPD
---------- Begin Solver Output -----------
Executing /home/neosotc/
  neos-5-solvers/snopt-ampl/snopt-driver.py
File exists
You are using the solver snopt.
Executing AMPL.
processing data.
processing commands.

2 variables, all nonlinear
1 constraint, all linear; 2 nonzeros
1 nonlinear objective; 2 nonzeros.

SNOPT 6.2-2: Optimal solution found expressions.
2 iterations, objective 4.5
```

The package also provides the function `neosSolveJob`, managing for submitting a job to the server, polling the queue until the job gets solved, and retrieving the result.

□ Training a Support Vector Machine

Support Vector Machines (SVM for short) [13] are models from the AI community used, besides other tasks, to assign patterns to two classes on the basis of a set of labeled examples $\{(x_1, y_1), \ldots, (x_m, y_m)\}$, where x_i is a vector and $y_i \in \{-1, +1\}$ identifies the corrisponding class. In its simplest form, the output of a SVM is the equation $w \cdot x + b = 0$ of a hyperplane gathering all patterns x_i such that $y_i = +1$ in one half−space, and the remaining patterns in the other half−space. This is achieved setting $w = \sum_{i=1}^{m} \alpha_i\, y_i\, x_i$ and $b = y_i - w \cdot x_i$ for i such that $\alpha_i > 0$, where α_i s are the solutions of the following quadratic constrained optimization problem:

$$\max \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i\, \alpha_j\, y_i\, y_j\, x_i \cdot x_j$$

$$\sum_{i=1}^{m} \alpha_i\, y_i = 0$$

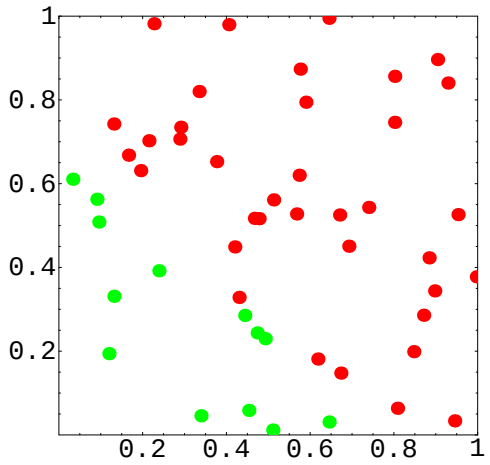
$$\alpha_i \geq 0 \quad \forall\ i = 1, \ldots m$$

The following function solves the above problem through the package described in the previous subsection, then scans the results in order to return a *Mathematica* expression gathering in a list the optimal values for α_i s:

```
svm[x_, y_] := Block[{r, i, k, retValue, m, n, alpha},
   m = Length[x]; n = Length[x[[1]]];
   r = "&lt;document&gt;\n";
   r = r <> "&lt;category&gt;nco&lt;/category&gt;\n";
   r = r <> "&lt;solver&gt;SNOPT&lt;/solver&gt;\n";
   r = r <> "&lt;inputMethod&gt;AMPL&lt;/inputMethod&gt;\n";
   r = r <> "&lt;model&gt;&lt;![CDATA[\n\n";
   r = r <> "param m integer > 0 default ";
   r = r <> ToString[m];
   r = r <> "; # number of sample points\n";
   r = r <> "param n integer > 0 default ";
   r = r <> ToString[n];
   r = r <> "; # sample space dimension\n\n";
   r = r <> "param x {1..m,1..n}; # sample points\n";
   r = r <> "param y {1..m}; # sample labels\n";
   r = r <> "param dot{i in 1..m,j in
       1..m}:=sum{k in 1..n}x[i,k]*x[j,k];\n\n";
   r = r <> "var alpha{1..m}>=0;\n";
   r = r <> "var w{1..n};\n\n";
   r = r <> "maximize quadratic_form:\n";
   r = r <> "sum{i in 1..m} alpha[i]\n";
   r = r <> "-1/2*\n";
   r = r <> "sum{i in 1..m,j in 1..m}
       alpha[i]*alpha[j]*y[i]*y[j]*dot[i,j];\n\n";
   r = r <> "subject to linear_constraint:\n";
   r = r <> "sum{i in 1..m} alpha[i]*y[i]=0;\n\n";
   r = r <> "]]&gt;&lt;/model&gt;\n";
   r = r <> "&lt;data&gt;&lt;![CDATA[\n\n";
   r = r <> "data;\n\n";
   r = r <> "param\tx:\t";
   For[k = 1, k ≤ n, k++, r = r <> ToString[k] <> "\t";];
   r = r <> ":=\n";
   For[i = 1, i ≤ m, i++, r = r <> ToString[i] <> "\t";
    For[k = 1, k ≤ n, k++,
     r = r <> ToString[x[[i]][[k]]] <> "\t";];
    r = r <> If[i == m, ";\n\n", "\n"];];
   r = r <> "param y :=\n";
   For[i = 1, i ≤ m, i++,
    r = r <> ToString[i] <> "\t" <> ToString[y[[i]]];
    r = r <> If[i == m, ";\n\n", "\n"];];
   r = r <> "]]&gt;&lt;/data&gt;\n\n";
   r = r <> "&lt;commands&gt;&lt;![CDATA[\n\n";
   r = r <> "option solver snopt;\n\n";
   r = r <> "solve;\n\n";
   r = r <> "printf: \"{\";\n";
   r = r <> "printf {k in 1..m-1}:\"%f,\",alpha[k];\n";
   r = r <> "printf: \"%f}\",alpha[m];\n";
   r = r <> "]]&gt;&lt;/commands&gt;\n\n";
   r = r <> "&lt;/document&gt;\n\n";
   retValue = neosAPI'neosSolveJob[r];
   alpha =
    StringReplace[retValue, ___ ~~ "{" ~~ o__ ~~ "}" ~~ ___ →
       "{" ~~ o ~~ "}"] // ToExpression;
   Return[alpha];
  ];
```

The use of this function is exemplified in the cells below, where the SVM algorithm is applied to a set of points labeled according to a fixed line. As a first step, 50 points are picked uniformly in the set $[0, 1]^2$, and each point is assigned to a half−space determined by a line chosen at random:

```
x0 = Random[];
x1 = Random[];
y0 = Random[];
y1 = Random[];
retta[xx_] = yy /. Flatten[Solve[
      (yy - y0) / (y1 - y0) == (xx - x0) / (x1 - x0), yy]] // Simplify;
grRetta = Plot[retta[xx], {xx, 0, 1}, PlotRange → {{0, 1}, {0, 1}},
   AspectRatio → Automatic, DisplayFunction → Identity];
m = 50; n = 2; x = Table[{Random[], Random[]}, {m}];
y =
  Table[If[x[[i]][[2]] - retta[x[[i]][[1]]] ≥ 0, 1, -1], {i, m}];
grPoints = Table[{AbsolutePointSize[5], RGBColor[(1 + y[[i]]) / 2,
      1 - (1 + y[[i]]) / 2, 0], Point[x[[i]]]}, {i, m}] // Graphics;
Show[grPoints, Axes → False, Frame → True,
  AspectRatio → Automatic];
```



Then, the original line is discarded, while the points' coordinates and labels are used as arguments of the `svm` function previously described.

```
alphas = svm[x, y]
```

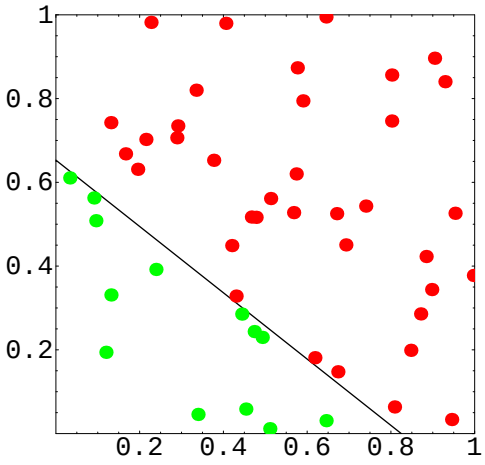
```
{2912.02, 0., 0., 0., 0., 0., 0., 0., 0., 223.635,
 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
 0., 0., 3135.66, 0., 0., 0., 0., 0., 0., 0., 0., 0.,
 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}
```

The return value is then used in order to obtain the parameters *w* and *b* of a line discriminating between the two classes of points, according to the formulas given at the beginning of this subsection:

```
w = Sum[alphas[[i]] y[[i]] x[[i]], {i, m}];
indices = Position[alphas, c_ /; c ≠ 0] // Flatten;
b = Mean[Table[y[[indices[[i]]]] - w.x[[indices[[i]]]],
    {i, Length[indices]}]];
```

Finally, the line is plotted and contrasted with the original set of points:

```
grHyp =
  ContourPlot[w.{xx, yy} + b, {xx, 0, 1}, {yy, 0, 1}, Contours → {0},
   ContourShading → False, DisplayFunction → Identity];
Show[grHyp, grPoints];
```



■ Acknowledgments

The author gratefully acknowledges partial support by the PASCAL Network of Excellence under EC grant no. 506778. This publication only reflects the author's views.

■ References

[1] R. Buyya (Ed.), *High Performance Cluster Computing: Architectures and Systems*. New York: Prentice Hall, 1999

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

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners−Lee, *Hypertext Transfer Protocol − HTTP/1.1*, RFC 2616, June 1999, ftp://ftp.isi.edu/in−notes/frc2616.txt

[4] I. Foster, C. Kesselman, *The grid: blueprint for a new computing infrastructure*. S. Francisco, CA: Morgan Kaufmann Publishers Inc., 1998

[5] International Organization for Standardization, ISO 8601 − Numeric representation of dates and time, 2003−01−30, http://www.iso.org/iso/en/prods−services/popstds/datesandtime.html

[6] S. Joseffson, *The Base16, Base32, and Base64 Data Encodings*, RFC 3548 http://www.ietf.org/rfc/rfc3548.txt

[7] D. Malchiodi, *A Mathematica Base64 package*, http://homes.dsi.unimi.it/~malchiod/software/base64, Jan 2006.

[8] D. Malchiodi, *A Mathematica xmlRpc package*, http://homes.dsi.unimi.it/~malchiod/software/xmlRpc, Jan 2006.

[9] D. Malchiodi, *Solving complex optimization problems through remote access to the NEOS server*, http://homes.dsi.unimi.it/~malchiod/software/neosAPI, Jan 2006.

[110] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*,Duxbury Press / Brooks/Cole Publishing Company, Second edition, 2002.

[11] J. Snell, D. Tidwell, P. Kulchenko, *Programming Web services with SOAP*. Sebastopol, CA: O'Reilly & Associates, 2002

[12] Sun Microsystem, Inc., RPC: Remote Procedure Call Protocol Specification, RFC 1050, http://www.faqs.org/rfcs/rfc1050.html, April 1998.

[13] C. Cortes and V. Vapnik, Support−Vector Networks, *Machine Learning*, 20 (1995), 121−167.

[14] D. Winer, *XML−RPC Specification*, June 15, 1999, http://www.xmlrpc.com/spec

[15] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg−McQueen, E. Maler, *Extensible Markup Language (XML) 1.0*, W3C Recommendation, Third Edition, February 4, 2004, http://www.w3.org/TR/2004/REC−xml−200402004