Embedding R within the Apache Web Server: What's the Use?

Jeffrey Horner

Abstract

The Apache web server is robust, scalable, and extensible. It is by far one of the most well-known, widely used and tested Open Source projects. With the 2.0 release came a powerful, enhanced module API that includes managing all stages of an HTTP request, an API for programming multi-processing modules (MPM), and new infrastructure to support serving multiple protocols, not just HTTP. Many popular scripting/interpreted languages such as Perl, Python, Ruby, and PHP have been implemented as Apache modules with great success. Applications utilizing these modules can be classified in two ways: those that serve dynamic content to web browsers, and those that serve dynamic content to other applications, e.g. web services.

This paper discusses the use of embedding R within the Apache web server. I will show how R users utilize the embedded R interpreter to create useful applications, both for web browsers and as web services. Further, I will compare this approach with other R projects involved in networked interfaces to R.

1 Introduction

The purpose of embedding a programming language inside the Apache 2.0 web server is the ability to write web applications in a programmer's language of choice. Some languages have lots of code libraries, like Perl and Python. Others were literally invented for programming web applications, like PHP. Still others are claimed as the "next best thing since sliced bread..." for web applications, such as Ruby on Rails. Regardless, choice is good.

But what about R? Are there any good reasons why a statistical language should be embedded into Apache 2.0? Perl [7], Python [6][9], and PHP[8] interface to R by linking to the R shared library. Ruby[14] has one in the works. One could use any of these to write a web application and then interface with R at the language level. However, one would have to know the language. Writing a web application in R allows statisticians to use a language they are familiar with, and eliminates the overhead of embedding the interface language run-time engine.

The R/Apache Integration project[5] was created to embed the R interpreter inside the Apache 2.0 web. It is composed of two parts. mod_R is the Apache

2.0 module that embeds the R interpreter, and RApache is the R package that interfaces R with Apache internals. The following discussion shows its use. Section 2 will lay out the R/Apache architecture, section 3 will explain how to write web applications with R/Apache, section 4 will discuss related works, and section 5 will conclude with some comments on future directions for the project.

2 Architecture

With version 2.0, the Apache developers created a highly customizable and portable software infrastructure[2]. Nearly every aspect of the web server is implemented as a module. Multi-processing modules (MPM) [3] were introduced to harness computer architecture dependent processing efficiencies. For instance, while forking a process on UNIX operating systems is cheap, this is not the case for Windows. Rather, threads are used for handling web requests and the ability to re-use sockets is also exploited. The mod_R module handles HTTP requests by handing them to the embedded R interpreter.

Porting Apache to various hardware architectures gave rise to the Apache Portable Runtime (APR) libraries [13]. APR functions range from memory management, file and directory management, network IO, to support for hash tables and string tables.

One important aspect of web application programming is the manipulation of the request data (or CGI data). This functionality is currently missing from the Apache 2.0 API and the APR, but is in fact managed in a sub-project, the libapreq2 library [11]. The RApache R package depends on libapreq2 to manage all the heavy lifting of the request data, so it will come bundled in the R/Apache source distribution until it's rolled into the API.

The following sections discuss where the mod_R module fits in with the MPM's and how the RApache packages takes advantage of APR and libapreq2.

2.1 Multi-Processing Modules

MPM's deal with making sure there are enough processes or threads laying idle to handle incoming web requests. They generally start one parent process and manage a collection of child processes, threads, or a combination of both. Figure 1 lists the four major MPM's currently supported by Apache 2.0.

The Worker, Perchild, and WinNT MPM's all implement threads in some way. Thus, they are currently unsuitable for embedding R. Worker starts one parent process which starts a variable number of children. Each child starts a listener thread and a fixed number of worker threads. As a web request comes it, one child process will respond. The listener thread dispatches the request to one of the worker threads. Worker's advantage is that it can scale rather quickly since one additional child can handle multiple requests. The disadvantage is that if one thread dies, the entire process dies, potentially taking down worker threads actively processing requests.

The Perchild MPM starts one parent process and a fixed number of child processes. Each child will start any number of worker threads as load dictates. The advantage of Perchild is that collections of children can be assigned a particular operating system user id. This is useful in virtual server environments where user specific data must not be seen across server boundaries, i.e. outside of the child process. The disadvantage is similar to Worker's, the potential of losing connections at a fast clip because of just one thread dying. Imagine if one virtual server had one dedicated child. If it goes down, all connections to the server are dropped until the parent process successfully starts another child.

The three previous MPMs all work on various UNIX style operating systems, but Windows gets an MPM all its own, WinNT. As mentioned above, Windows processes are heavy-weight, meaning they are costly to spawn. Thus the WinNT parent process starts only one child process with many, many threads.

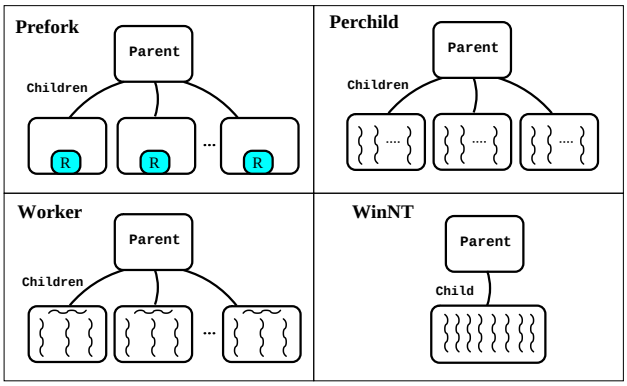

Figure 1: Apache 2.0 Multi-Processing Modules.

The Prefork MPM implements one parent process managing a collection of child processes; no threads are involved. It is functionally equivalent to the original Apache 1.3 architecture, and it is also the MPM needed to run mod_R. As web requests come in, one child is chosen to serve the request, and the others are kept idle. The parent always keeps around a few idle children to manage potential load spikes. Thus, preforking hides the cost of forking new child processes which must potentially initialize language interpreters, as is the case for mod_R.

2.2 The mod_R Module

When Apache is built from source files, one MPM is chosen to be statically compiled into the server. Any number of other types of modules can either be statically compiled into the server or loaded dynamically at runtime. This discussion presumes that the mod_R module will be loaded at runtime. Consider the following Apache configuration as it pertains to mod_R (refer to [12] for a

detailed explanation of the Apache 2.0 configuration files):

```
LoadModule R_module /usr/lib/apache2/modules/mod_R.so

<Location /test/R>
    SetHandler r-handler
    Rlibrary GDD
    Rsource /var/www/html/test.R
    RreqHandler handler
</Location>
```

To paraphrase the above, the web server first loads the mod_R module located at `/usr/lib/apache2/modules/mod_R.so`, looks for the special module structure variable `R_module`, and uses it to initialize the embedded R interpreter. Then, on the first incoming HTTP request for `http://localhost/test/R`, mod_R will execute:

```
library("GDD")
source("/var/www/html/test.R")
handler(r)
```

and on subsequent requests it will execute `handler(r)` only. The special request record variable `r` is described later in section 2.3.

Apache gives each module a chance to handle all incoming requests, even if the modules has not been configured to do so. The `SetHandler` Apache directive can be used to tag requests with a string value. Thus, for all incoming requests mod_R will check if the handler tag is set to `r-handler` and handle it. Otherwise it declines.

The mod_R directives `Rsource` and `Rlibrary` are used to load R code into the embedded R interpreter. As described above, the code is not loaded until the first request comes in. This allows the server to start up quickly and to spawn new children quickly as server load increases. The disadvantage is that a developer may not know until request time if her/his code contains any errors. Each directive must appear only once within any `<Location>` or `<Directory>` directive. This restriction may change in future versions.

The mod_R directive `RreqHandler` specifies an R function to use for handling requests. The function should be defined in one of the files or packages loaded by the directives above. Otherwise, mod_R won't know about it. It also must appear only once within any `<Location>` or `<Directory>` directive.

Another important component of the Apache module design, and a free benefit to mod_R, is its use of APR's memory pools. The APR implements a crafty memory management scheme around the idea of a pool of memory. When the pool is destroyed, all the memory ever allocated out of it is also destroyed, or `free()`'d. This is important as it relates to request processing. Each request that comes in is attached to a new memory pool. Once the request has been handled, the request is destroyed along with its pool, warding off any memory leaks that could potentially happen.

2.3 The RApache Package

Once mod_R has handed off the request to a function like `handler()`, the RApache package takes charge of Apache related data and tasks. Since it is so essential, mod_R implicitly loads it, thus one never needs to call `library(RApache)`.

The function `handler` takes one argument, an RApache request record based on the Apache structure `request_rec`. It contains everything a programmer needs to know about the incoming HTTP request. RApache implements the request record as an R external pointer of class `ra_request_rec` and is treated similarly to an R list. Each list element is a character vector of length one; the values are described later. Both the `$` and `[[` indexing operators are available, however partial completion of element names is not performed. Many of the elements are simple types, either character vectors, integer vectors, or POSIXct objects, all of length one. Other elements such as `headers_in` and `headers_out` are of type `apr_table`.

The APR provides the apr_table type, which RApache implements as an R external pointer. It is also treated similarly to an R list whose elements and values are character vectors. The `$` and `[[` indexing operators are also available and again, partial completion of element names is not performed. Dispatching is done S3 style on the class attribute `apr_table`, and each element of an `apr_table` is case-insensitive.

RApache also relies on libapreq2 for parsing CGI data, which is built on the APR. This includes parsers for GET arguments, POST data (which includes `multipart/mixed` and `application/x-www-form-urlencoded` encodings), and HTTP Cookie headers. libapreq2 turns each of these data points into variables of type `apr_table`, which RApache in turn converts into R external pointer types for R consumption.

3 Application Design

Writing web applications with mod_R and RApache is best summed up in the *hello world* example:

```
handler <- function(r){
    apache.write(r,"<h1>Hello World!</h1>")
    OK
}
```

which produces the output in Figure 2.

The author presumes the reader has a basic understanding of web application development and HTTP request and response messages. Please see [1] for a beginner's style introduction to HTTP or see [4] for the HTTP 1.1 specification.

The *hello world* example illuminates three key parts of mod_R and RApache web applications:

- it is defined by an R function which takes one argument, the RApache request record,

Figure 2: Output of *hello world* handler.

- it utilizes the RApache package functions, such as `apache.write()`,
- and it returns a value from the available set of RApache return codes.

One can write pretty sophisticated web applications with the current version of mod_R and RApache, far beyond the *hello world* example. The APPENDIX provides two other example handlers for the reader to peruse, with source code and output screen shots.

3.1 The RApache request record

As stated in the previous section, the RApache request record contains everything a programmer needs or wants to know about the incoming request. Its list elements are:

`headers_in`: An object of class `apr_table` containing all the HTTP headers sent by the client.

`headers_out`: An object of class `apr_table` containing all the HTTP headers to send to the client.

`err_headers_out`: object of class `apr_table` containing headers to be sent when an error occurs; they persist across internal redirects.

`proto_num`: Integer. Protocol version number of protocol; 1.1 = 1001.

`protocol`: Character. Protocol string, as given to us, or HTTP/0.9.

`unparsed_uri`: Character. The URI without any parsing performed.

`uri`: Character. The path portion of the URI.

`canonical_filename`: Character. The true filename, we canonicalize r$filename if these don't match.

`path_info`: Character. The PATH_INFO extracted from this request.

`args`: Character. The QUERY_ARGS extracted from this request.

`content_type`:Character. The content-type for the current request.

`handler`: Character. Should always be r-handler. The handler string that we use to call a handler function.

`content_encoding`: Character. How to encode the data.

`range`: Character. The `Range:` header.

`hostname`: Character. The server hostname.

`user`: Character. If an authentication check was made, this gets set to the user name.

`header_only`: Logical. HEAD request, as opposed to GET.

`no_cache`: Logical. This response can not be cached.

`no_local_copy`: Logical. There is no local copy of this response.

`status`: Integer. Status line.

`method_number`: Integer. M_GET, M_POST, etc. See the RApache documentation for more details.

`eos_sent`: Logical. A flag to determine if the eos bucket has been sent yet.

`the_request`: Character. First line of the request.

`method`: Character. Request method (eg. GET, HEAD, POST, etc.)

`status_line`: Character. Status line, if set by script.

`bytes_sent`: Numeric. Number of bytes sent.

`clength`: Numeric. The "real" content length.

`remaining`: Numeric. Remaining bytes left to read from the request body.

`read_length`: Numeric. Number of bytes that have been read from the request body.

`request_time`: POSIXct DateTime object. Time when the request started.

`mtime`: POSIXct DateTime object. Last modified time of the requested resource.

3.2 RApache package functions

These functions implement the basics of reading and writing HTTP request and response messages. See [5] for more complete documentation.

`apache.read` and `apache.readline`: These two functions are used to read in the HTTP request data. They are useful for constructing web service style applications which consume their messages as XML documents.

`apache.get_post` and `apache.get_uploads`: These two functions are also used to read in the HTTP request data. However, they rely on libapreq2 to take control and parse the entire incoming request data, so they cannot be used if `apache.read` or `apache.readline` were already called. `apache.get_post` will return all the HTTP `POST` form data as an object of type `apr_table`, explained previously. `apache.uploads` returns an R list of all the files uploaded when the `POST` data was encoded as `multipart/form-data`. Each list element is itself a list containing the original name of the uploaded file and the temporary location on the server where the file was uploaded. Programmers can then copy that temporary file to their desired location since it will be deleted when the request has been processed.

`apache.write`: Write data to the outgoing HTTP response message, i.e. the browser.

`apache.get_args`: This function also uses libapreq2 to parse the HTTP `GET` data. The return value is an object of class `apr_table`.

`apache.get_cookies` and `apache.add_cookies`: Functions for getting and setting HTTP cookies.

`apache.add_header`: Adds headers to the HTTP response message.

`apache.log_error`: Writes messages to the Apache configured error log file.

`apache.set_content_type`: Sets the content type.

3.3 RApache return codes

Every handler must end by returning an RApache return code from one of two sets. The first set is related to how the handler actually handles the request. They are:

DONE The handler has *completely* handled the HTTP request, and no other Apache internal request phases should alter it.

DECLINED The handler decided not to do anything to the request, thus Apache will satisfy it in some other way.

OK The handler has done its part with the request, and it's okay for Apache to send it on to the other request phases. *In general, handlers should return this code.*

The second set of codes are the actual valid HTTP response codes. See [4] for more information. They are:

```
HTTP_ACCEPTED                HTTP_NOT_IMPLEMENTED
HTTP_BAD_GATEWAY             HTTP_NOT_MODIFIED
HTTP_BAD_REQUEST             HTTP_OK
HTTP_CONFLICT                HTTP_PARTIAL_CONTENT
HTTP_CONTINUE                HTTP_PAYMENT_REQUIRED
HTTP_CREATED                 HTTP_PRECONDITION_FAILED
HTTP_EXPECTATION_FAILED      HTTP_PROCESSING
HTTP_FAILED_DEPENDENCY       HTTP_PROXY_AUTHENTICATION_REQUIRED
HTTP_FORBIDDEN               HTTP_RANGE_NOT_SATISFIABLE
HTTP_GATEWAY_TIME_OUT        HTTP_REQUEST_ENTITY_TOO_LARGE
HTTP_GONE                    HTTP_REQUEST_TIME_OUT
HTTP_INSUFFICIENT_STORAGE    HTTP_REQUEST_URI_TOO_LARGE
HTTP_INTERNAL_SERVER_ERROR   HTTP_RESET_CONTENT
HTTP_LENGTH_REQUIRED         HTTP_SEE_OTHER
HTTP_LOCKED                  HTTP_SERVICE_UNAVAILABLE
HTTP_METHOD_NOT_ALLOWED      HTTP_SWITCHING_PROTOCOLS
HTTP_MOVED_PERMANENTLY       HTTP_TEMPORARY_REDIRECT
HTTP_MOVED_TEMPORARILY       HTTP_UNAUTHORIZED
HTTP_MULTIPLE_CHOICES        HTTP_UNPROCESSABLE_ENTITY
HTTP_MULTI_STATUS            HTTP_UNSUPPORTED_MEDIA_TYPE
HTTP_NO_CONTENT              HTTP_UPGRADE_REQUIRED
HTTP_NON_AUTHORITATIVE       HTTP_USE_PROXY
HTTP_NOT_ACCEPTABLE          HTTP_VARIANT_ALSO_VARIES
HTTP_NOT_EXTENDED            HTTP_VERSION_NOT_SUPPORTED
HTTP_NOT_FOUND
```

3.4 The State Problem

One caveat to programming with mod_R and RApache is the fact that no data are shared between the multiple instances of R interpreters embedded in each Apache process. This caveat exists in other embedded languages as well. For instance, if one handler stuffs R variables into the R Global Environment during one web request, depending on which Apache process is chosen, that variable may or may not exist upon the next web request.

Using HTTP Cookie headers or HTTP GET variables solves this problem for small amounts of data, and while there are solutions for storing and retrieving larger datasets, concurrency issues must be addressed. Using SQL Databases provides one solution, but some statistical data sets are not amenable to database storage. Another solution is to use the R functions `load` and `save`, or similar functions that read data from a file or connection, however one or more web requests accessing or storing the file at the same time may cause corruption.

The author intends to research this problem for future versions of mod_R and RApache.

4 Related Works

4.1 RSOAP

RSOAP is part of the larger project RStatServer [17], which utilizes RSOAP on the back end to provide a total statistical web application experience. *RSOAP* is a web server of sorts that provides an XML SOAP interface to R. Web clients must construct and consume SOAP encoded messages. RSOAP does not exhibit the state problem of mod_R. When a new web client connects, RSOAP spawns a new process that handles each and every client request.

RSOAP is best used as a web services style interface to R, while mod_R is more general purpose. With the RApache package, programmers can construct either traditional style web applications for web browsers or web services style applications like RSOAP.

4.2 RServe

Rserve provides an interface to R using the traditional client/server paradigm. Clients written in either Java or C/C++ communicate with an *Rserve* server via a custom protocol built on top of TCP/IP. Clients need not worry about initializing an R session or linking to the R library. Like *RSOAP*, it forks a new process for each new client connection, thus the state problem is mitigated.

In order to utilize *Rserve* in a web application, one must deploy a Java application server such as Tomcat or JBoss, or write CGI programs in C/C++.

4.3 Other Related Works

There are several other projects related to web programming with R. They are:

- CGIwithR
  http://www2.warwick.ac.uk/fac/sci/statistics/staff/academic/firth/software/cgiwithr/
  Allows files containing R code to be treated as CGI scripts. Each HTTP request must start a new R interpreter.

- R_PHP_Online
  http://steve-chen.net/R_PHP/
  Allows R code to be submitted to a PHP script which in turn executes the R interpreter. Each HTTP request must start a new R interpreter.

- Rho
  http://www.rho-project.org/
  A web-based Java framework for R analysis.

- Rpad
  http://www.rpad.org/Rpad/
  A web based analysis program. Users interact with R through a customized web page. For single user intractive use, a TCL web server is

used. Multi-user web based support is available via a Perl interface either on Apache or IIS. One R interpreter is created to serve all requests.

- Rweb
  http://www.math.montana.edu/Rweb
  A set of Perl CGI scripts to facilitate a web based interface to R. Again, each request must start a new R interpreter.

5 Conclusion and Future Work

In its current state, the R/Apache project offers a complete framework for creating many CGI style and web services style applications written exclusively in R. No interface language or decoupled networked interface to R is needed. Also, R/Apache could allow projects like Rserve[15] and RSOAP[17] to leverage Apache's very robust and stable networking code base which includes support for multi-processing modules and even custom protocols.

The first area that can use some exploration is how to maintain state in the R/Apache framework. One avenue that looks promising is the use of the memcached project (`http://www.danga.com/memcached/`), a distributed memory object caching system. This has been used successfully in large web applications to decrease database load by caching information related to state management. It's conceivable that R objects may be stored and retrieved from a memcached server. Another option is to write a custom Apache MPM based on Prefork that implements sessions. A session would allow a web client to communicate exclusively with the same Apache child on each request, allowing for state to be maintained for the duration of the session. Other areas to explore are database and file connections.

References

[1] Http: From wikipedia, the free encyclopedia. URL `http://en.wikipedia.org/wiki/HTTP`.

[2] Ryan B. Bloom. *Apache Server 2.0: The Complete Reference*. McGraw-Hill/Osborne, Berkeley, California, 2002.

[3] Ryan B. Bloom. *Multi-Processing Modules*, chapter 7, pages 129–160. In [2], 2002.

[4] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. IETF RFC 2616: Hypertext transfer protocol – HTTP/1.1. Web Page, June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`.

[5] Jeffrey Horner. The R/apache integration project, 2005. URL `http://biostat.mc.vanderbilt.edu/twiki/bin/view/Main/ApacheRproject`.

[6] Duncan Temple Lang. R/splus-python interface, 2004. URL http://www.omegahat.org/RSPython/.

[7] Duncan Temple Lang. The R/splus - perl interface, 2005. URL http://www.omegahat.org/RSPerl/.

[8] Dieter Menne. phpserialize: Serialize R to php associative array, 2005. URL http://cran.r-project.org/src/contrib/Descriptions/phpSerialize.html.

[9] Walter Moreira and Gregory R. Warnes. Rpy (R from python), 2005. URL http://rpy.sourceforge.net/.

[10] Francois Pinard. Nrart newton-raphson art. URL http://pinard.progiciels-bpi.ca/plaisirs/animations/NRart/NRart.html.

[11] Apache HTTP Server Project. Apache http request library. URL http://httpd.apache.org/apreq/.

[12] Apache HTTP Server Project. Apache http server version 2.0 documentation. URL http://httpd.apache.org/docs/2.0/.

[13] Apache HTTP Server Project. Apache portable runtime project. URL http://apr.apache.org.

[14] Kaspar Schiess. R 4 ruby library, 2004. URL http://r4ruby.rubyforge.org/wiki/wiki.pl.

[15] Simon Urbanek. Rserve – a fast way to provide R functionality to applications. In Kurt Hornik and Friedrich Leisch, editors, *DSC 2003 Proceedings of the 3rd International Workshop on Distributed Statistical Computing*, 2003. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

[16] Simon Urbanek. Gdd: R package, 2004. URL http://www.rosuda.org/R/GDD/.

[17] Gregory R. Warnes. Rstatserver project. URL http://research.warnes.net/projects/RStatServer/.

APPENDIX

A: The Test handler

This handler shows a more complete example of how to utilize the RApache functions to construct web applications:

```
handler <- function(r)
{
    # Grab all incoming HTTP request data
    args <- apache.get_args(r)
    post <- apache.get_post(r)
    cookies <- apache.get_cookies(r)
    uploads <- apache.get_uploads(r)

    # Test if the GET variable called was set
    called <- if(length(args$called))
        as.integer(args$called) + 1
    else
        called <- 1

    # Set a cookie with the incremented called value.
    # expires in 100 seconds.
    apache.add_cookie(r, "called", called, expires=Sys.time()+100)

    apache.write(r,"<HTML><BODY><H1>Hello from mod_R</H1>")

    # Write out the form for capturing GET,POST, and file uploads
    apache.write(r,
        '<form enctype=multipart/form-data method=POST action="/test/R?called=',
        called,'">',
        sep="")
    apache.write(r,
        'Enter a string: <input type=text name=name value="',
        post$name,'"><br>',
        sep="")
    apache.write(r,
        'Enter another string: <input type=text name=name value="',
        post$name,'"><br>',
        sep="")
    apache.write(r,
        'Upload a file: <input type=file name=file><br>')
    apache.write(r,"<input type=submit name=Submit>")

    # Now write out everything we saw
    apache.write(r,'<hr>')
    apache.write(r,"<h2>Args variables</h2>")
    apache.write(r,as.html(args));
    apache.write(r,"<h2>Post variables</h2>")
    apache.write(r,as.html(post))
```

```
    apache.write(r,"<h2>Cookies</h2>")
    apache.write(r,as.html(cookies))
    apache.write(r,"<h2>File Uploads</h2>")
    apache.write(r,as.html(uploads))
    apache.write(r,"<h2>Request Record</h2>")
    apache.write(r,as.html(r))
    apache.write(r,"</BODY></HTML>")

    OK
}
```

The output spans Figure 3 and 4.

B: The GD/NRart handler

Here's another handler: an example of how mod_R and RApache can produce dynamic plots. Unfortunately, it requires a custom modification to the R package GDD [16]. It also uses NRart [10].

```
library(GDD)
library(NRart)
r2 <- function(r)
{
    step <- 2
    args <- apache.get_args(r)

    if(length(args$t)){
        pstep <- as.integer(args$t)
        if (pstep > step && pstep < 121)
            step <- pstep;
    }

    apache.set_content_type(r,"image/png")

    GDD(ctx=apache.gdlib_ioctx(r),w=500,h=500,type="png")

    nr.movie(x^3 + .28 * tan(x + t) + cos(x + 2*t)*.3i - 0.7,
                't', seq(0, pi, length=121)[step],
                extent=1, steps=3, points=400,
                col=rainbow(256), zlim=c(-pi, pi))

    dev.off()
    OK
}
```

The Apache configuration file looks like this:

```
<Location /dynamic/plot.png>
        SetHandler r-handler
        Rsource /home/hornerjr/R_MODULE/mod_R/test/test2.R
        RreqHandler r2
</Location>
```

See Figure 5 to view the output.

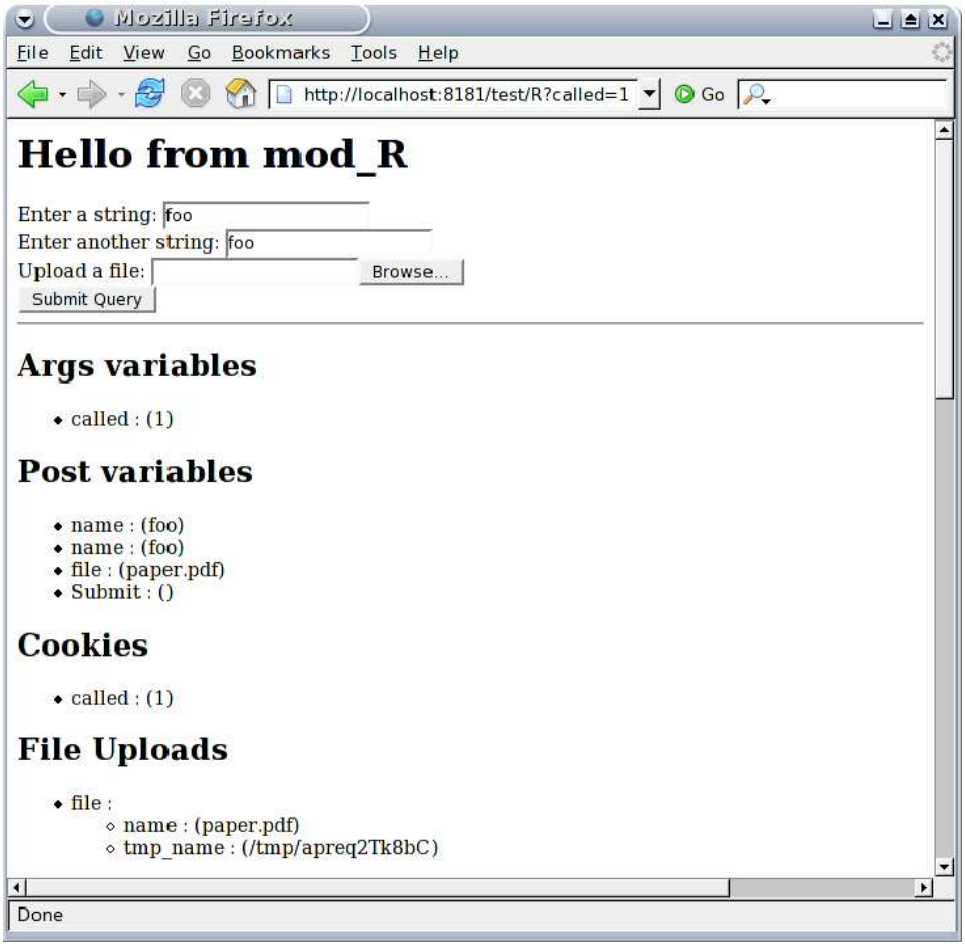

Figure 3: Top part of output from the test handler.

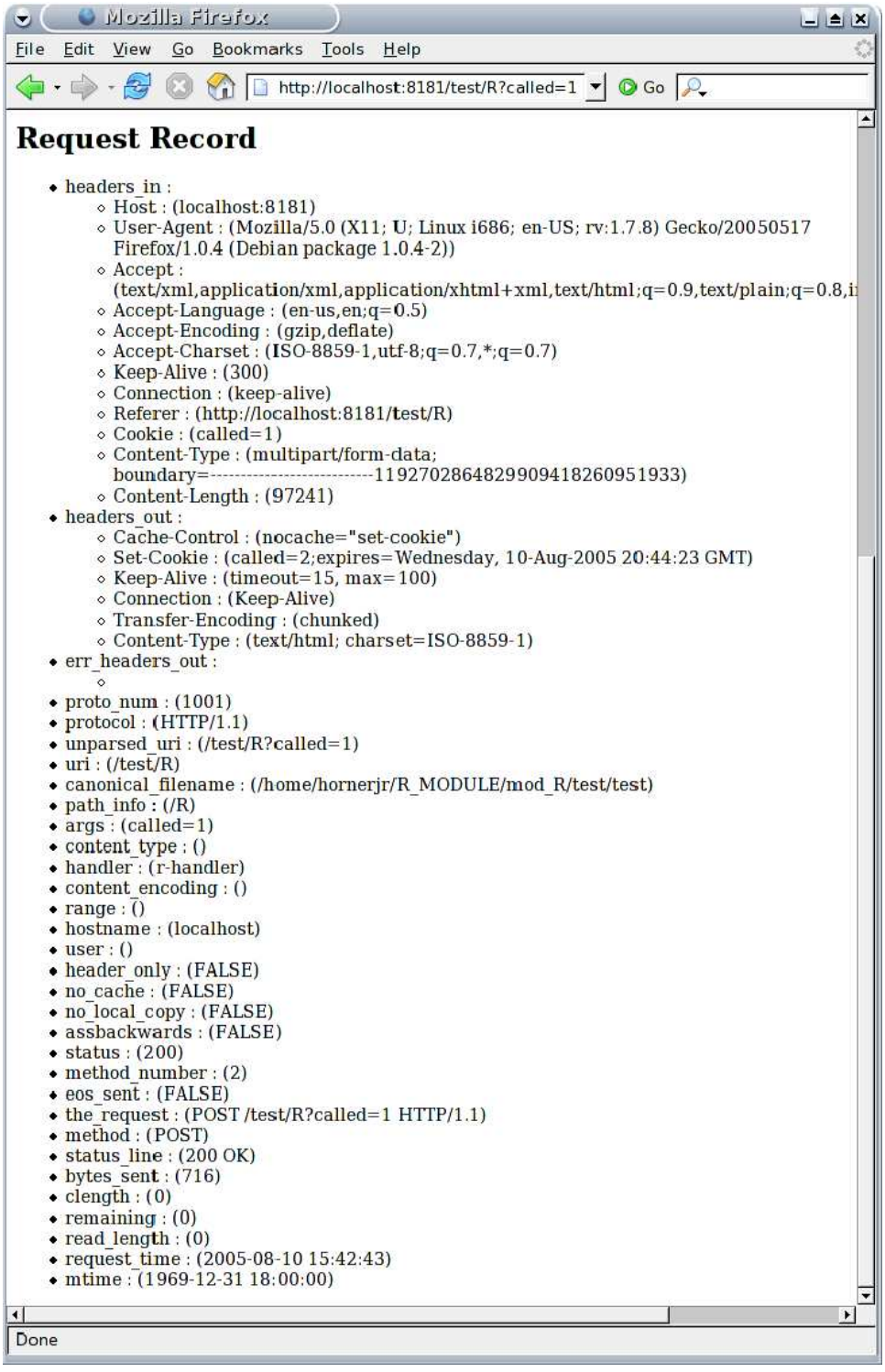

Figure 4: Bottom part of output from the test handler.

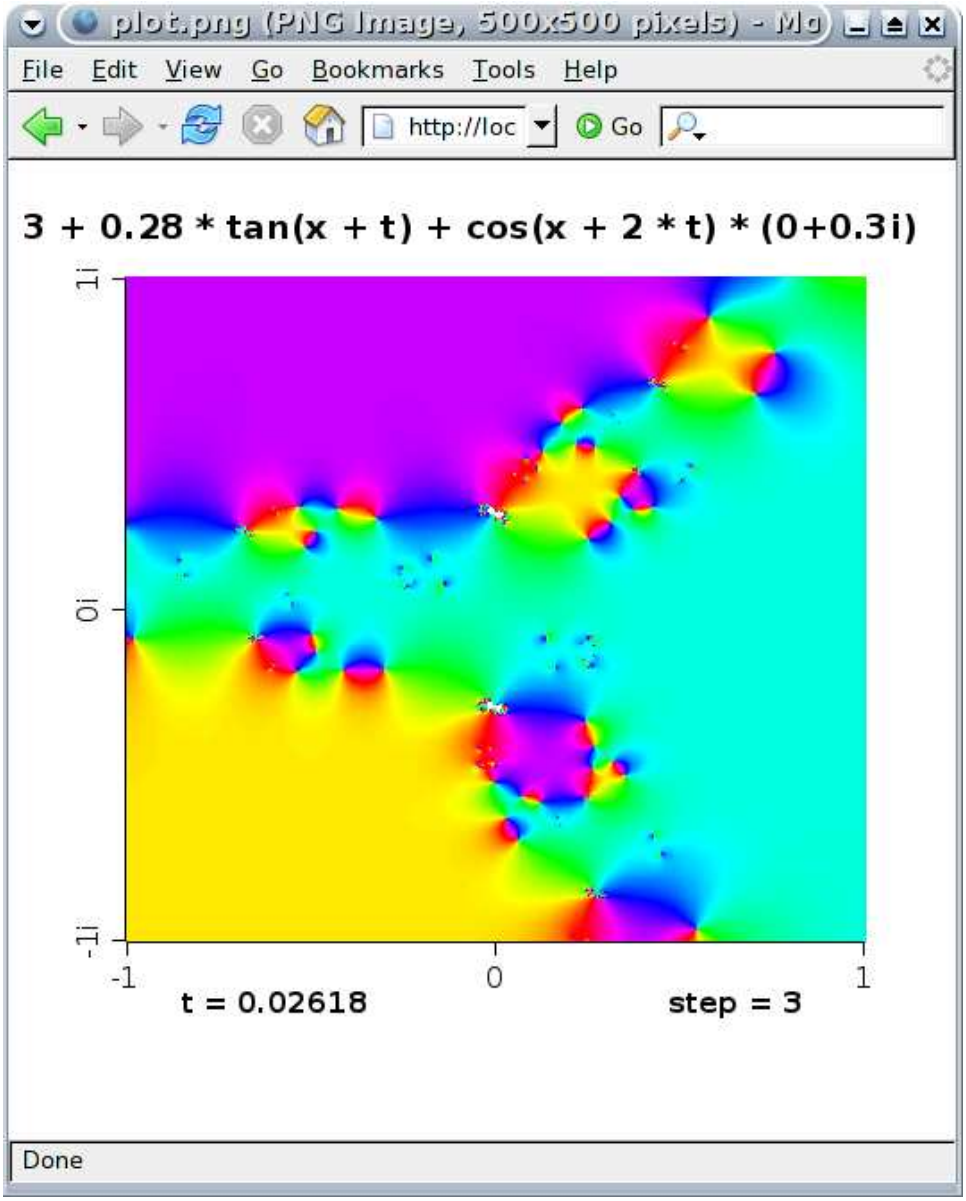

Figure 5: Output from GDD/NRart handler.