Perl for the working (or aspiring) mathematician

John Kerl

Department of Mathematics, University of Arizona
Software Interest Group

October 11, 2006

Abstract

What can one more language do for you?

- Why people use Perl; why some people love it and why others hate it.
- The minimum you need to know to write and run simple Perl programs, even if you haven't used such languages before.
- Text manipulation and regular expressions.
- Numerics.

Why Perl?

- Perl programs are fast to develop: no compilation phase.
- Perl is the de facto standard for CGI scripting.
- Unlike monolithic programs such as Matlab, Mathematica, etc., it's easy to connect (pipe) Perl scripts to other programs.
- Perl has built-in support for text manipulation.
- It works the same on Unix/Linux or Windows. (In fact, this is why I started using Perl back in 2000.)

- It's not the best language for doing math, but it's good for quick-and-dirty programs. Perl does math as well as many other things (a general-purpose language).
- Perl is good for automating all those miscellaneous tedious time-consuming tasks — times when you think *I am really sick and tired of this miscellaneous tedious time-consuming task which is keeping me from what I'd rather be doing* — grade calcuations, turning data into nicely formatting LaTeX output, making bulk changes to many files at once, making a nice index for LaTeX documents, etc. etc.

Love/hate

Perl motto: There's more than one way to do it (TMTOWTDI). Larry Wall took favorite features from many languages and put them into Perl. It's powerful and maddeningly eclectic.

Reading proficiency

The department has a **foreign language** requirement for PhD students. Why? Well, not all important results are published in your native language!

By the same token, if you're doing compuational work of any sort, you may find that your collaborators' favorite programming languages aren't the same as yours.

Learn some Python, Perl, C, Fortran, Matlab.

Magic

There are many pre-existing software tools. In this presentation, I'll give examples of an ODE solver and a data formatter. Matlab et al. already do these things for you. Why should you listen to me reinventing the wheel?

- Pre-existing packages don't always have what you want — be ready to roll your own when necessary.
- Computers appear to be magical. When you learn to program, you learn to create your own wizardry and find out how things really work. Perl is a good starter language. (Python is another good choice!)

Installation

Unix/Linux: It's already there.

Mac: It's there too — Perl is included in Mac OS X.

Windows: Download it from the web.

Running Perl programs

Put the following text in a file, say `hello.pl`:

```
print "Hello, world!\n";
```

Then at the command prompt, type:

```
[shell prompt]$ perl hello.pl
```

You should see:

```
Hello, world!
```

More about running Perl programs

For simplicity of presentation, I suggested typing `perl hello.pl`.

You can also just type `hello.pl`: Use `#!` (shebang) for Unix/Linux; `.pl` association for Windows. (The `.pl` suffix is a mere convention for Linux, and can matter on Windows.) I can give you more info about this topic but it's a technical detail.

Numerics by example

Here is a naive Euler (first-order) integrator for the Lorenz equations. (Feel free to implement higher-order techniques such as Runge-Kutta — it would simply occupy another slide or two.)

Derivation of Euler scheme:

$$\begin{aligned} x &= x(t) \\ x' &\approx \frac{x(t+h) - x(t)}{h} \\ x(t+h) &\approx x(t) + hx' \\ x_{k+1} &:= x_k + hx' \end{aligned}$$

Let's turn that into instructions for a computer to carry out, using the Lorenz equations.

Lorenz equations

System of non-linear ODEs:

$$\begin{aligned} x' &= s(y-x) \\ y' &= rx - y - xz \\ z' &= xy - bz. \end{aligned}$$

Constants:

$$s = 10.0, \qquad r = 28.0, \qquad b = 2.6666667.$$

Initial conditions:

$$x_0 = 10, \qquad y_0 = 0, \qquad z_0 = 10.$$

First half of numerical script

```
$tmax = 10.0;                    # Simulate 10 seconds
$h    = 0.002;                   # with step of 0.002 sec
$N    = int($tmax/$h);

$x0 = 10; $y0 = 0; $z0 = 10;    # Initial conditions

$s = 10.0; $r = 28.0; $b = 2.666667; # Constants
```

Second half of numerical script

```
$x = $x0; $y = $y0; $z = $z0;  # Start integration
for ($t = 0; $t < $tmax; $t = $t + $h) {
    printf "%11.8f\t%11.8f\t%11.8f\n", $x, $y, $z;

    $xp = $s * ($y - $x);      # Cheesy Euler integrator.
    $yp = $r * $x - $y - $x * $z;
    $zp = $x * $y - $b * $z;

    $x = $x + $h * $xp;
    $y = $y + $h * $yp;
    $z = $z + $h * $zp;
}
```

Language points

- Scalar (think "non-array") variables begin with a dollar sign.
- Variable assignments, arithmetic expressions, etc. are about what you'd expect.
- Statements are terminated by semicolons.
- Control structures (for, if, while, etc.) are delimited using curly braces.

Output of numerical script

```
[shell prompt]$ perl lor2.pl
10.00000000  0.00000000 10.00000000
 9.80000000  0.36000000  9.94666666
 9.61120000  0.71312533  9.90067376
 9.43323851  1.05961157  9.86157814
 9.26576597  1.39970047  9.82897419
 9.10844466  1.73363801  9.80249158
 8.96094853  2.06167273  9.78179311
 ...         ...         ...
```

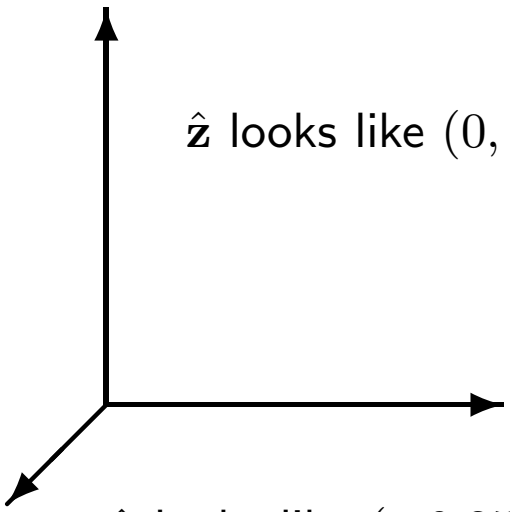
OK, so we're seeing the estimated x's, y's, and z's. But how to plot three variables on a two-dimensional screen?

Text manipulation by example

Answer: Projection from 3D to 2D is just a linear transformation. Write down a 2x3 matrix P whose columns are where you want $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ to go. Pre-multiply each data triple by that:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (P\hat{\mathbf{x}})_0 & (P\hat{\mathbf{y}})_0 & (P\hat{\mathbf{z}})_0 \\ (P\hat{\mathbf{x}})_1 & (P\hat{\mathbf{y}})_1 & (P\hat{\mathbf{z}})_1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Then plot (u, v) pairs on the two-dimensional display.



$\hat{\mathbf{z}}$ looks like $(0, 1)$ in 2D

$\hat{\mathbf{y}}$ looks like $(1, 0)$ in 2D

$\hat{\mathbf{x}}$ looks like $(-0.25, -0.25)$ in 2D

$$P = \begin{bmatrix} -0.25 & 1 & 0 \\ -0.25 & 0 & 1 \end{bmatrix}$$

Outline of projection script

We'll make a separate (re-usable!) script that simply reads three-column data — from our Lorenz integrator *or anything else* — and writes two-column data, modified as described above:

- **Read** lines of input (triples of numbers, delimited by whitespace) one line at a time, splitting the line into three numbers x, y, and z.
- For each triple x, y, z, **compute** u and v.
- **Write** u and v.

Subroutines

Here's the computation part. I'll use this as an example of subroutines. Assume we've already read in a triple of x, y, and z. We'll call this routine to find what u and v are.

```
@Pxhat = (-0.25, -0.25); @Pyhat = (1, 0); @Pzhat = (0, 1);
sub do_triple
{
    my ($x, $y, $z) = @_; # Use "my" to keep them local &
    my ($u, $v);          #   not clash w/ outer x, y, z.
    $u = $Pxhat[0]*$x + $Pyhat[0]*$y + $Pzhat[0]*$z;
    $v = $Pxhat[1]*$x + $Pyhat[1]*$y + $Pzhat[1]*$z;
    return ($u, $v);
}
```

Language points

- Subroutines start with the `sub` keyword, and they return values (scalars, arrays, etc.) using the `return` keyword.
- The argument list is spelled `@_`. Keep local variables private to your subroutine using `my`.
- You can have an array of scalars on the left-hand side of an assignment.
- Arrays are indexed (from zero up, math folks!) using square brackets.

Reading lines of standard input

```
$lno = 0;               # Remember line number so we can
while ($line = <>) {  # print informative error messages.
    chomp $line;
    $line =~ s/^\s+//; # Strip leading whitespace.
    $line =~ s/\s+$//; # Strip trailing whitespace.
    $lno++;
    @flds = split /\s+/, $line;
    $num_flds = @flds;
    if (@flds == 3) {
        ($u, $v) = do_triple($flds[0], $flds[1], $flds[2]);
        printf "%11.7f %11.7f\n", $u, $v;  # Output
    } else {
        die "$0:  Need 3 columns; got $n at line $lno.\n";
    }
}
```

Language points

- <> on the right-hand side of an assignment gets the next line of input.
- `s/old/new/` is a substitution. The "old part" is a **regular expression**. Here, `s/^\s+//` means replace one or more whitespace characters at the beginning of the line with nothing.
- The `split` keyword splits a string into pieces, with a delimiter specified by a regular expression of your choice. It hands back an array of pieces.
- The `printf` syntax (along with `for`) is taken straight from C.

Command-line arguments to the projection script

```
@Pxhat = (-0.25, -0.25); @Pyhat = (1, 0); @Pzhat = (0, 1);
while (@ARGV && ($ARGV[0] =~ m/^-/)) {
    if ($ARGV[0] eq "-xy") {
        @Pxhat = (1, 0);
        @Pyhat = (0, 1);
        @Pzhat = (0, 0);
    }
    elsif (...) {
        ...
    }
    else {
        last
    }
    shift @ARGV;
}
```

Language points

- Command-line arguments to a Perl script are useful for allowing the user (you!) to specify initial values, options, etc. instead of typing all parameters into the script.
- `@ARGV` is an array of the command-line arguments to the script. (E.g. `perl projdown.pl -xy`.)
- `m/pattern/` is another use of regular expressions: here, for matching. Take the specified action if the command-line argument begins with a minus sign.
- `shift` removes the zeroth element of an array, shortening it by one.

Output of numerical and projection scripts

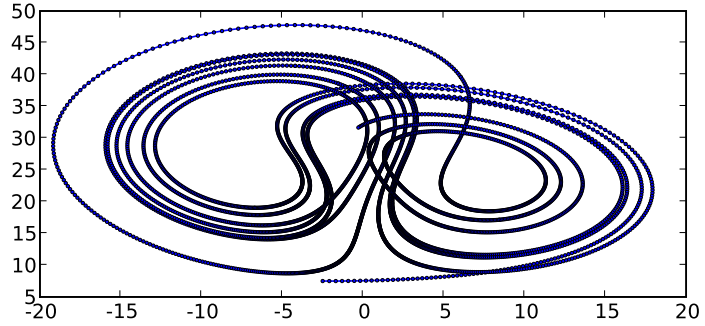
```
[shell prompt]$ perl lor2.pl | perl projdown.pl
 -2.5000000   7.5000000
 -2.0900000   7.4966667
 -1.6896747   7.4978738
 -1.2986981   7.5032685
 -0.9167410   7.5125327
 ...          ...
```

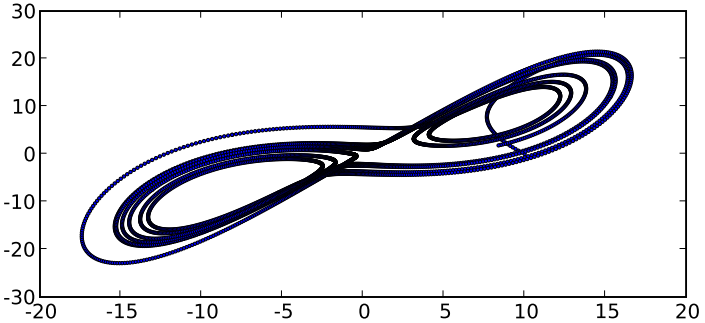
Or:

```
[shell prompt]$ perl lor2.pl > xyz.txt
[shell prompt]$ perl projdown.pl < xyz.txt > uv.txt
```

These data can be sent to a 2D display. (See next slide.)

Output of numerical and projection scripts





Also here are the full scripts:

`http://math.arizona.edu/~kerl/doc/perl-talk/lor2.pl.txt`

`http://math.arizona.edu/~kerl/doc/perl-talk/projdown.pl.txt`

More text processing

Some raw data (a 3x3 matrix):

```
0.8550000   0.1900000  -0.0450000
0.0550000  -0.2100000   1.1550000
3.2550000  -8.6100000   6.3550000
```

Run it through the Perl program `mattex` (available at `http://math.arizona.edu/~kerl`):

```
[shell prompt]$ mattex myfile.txt
\left[\begin{array}{rrr}
0.8550000 &  0.1900000 & -0.0450000 \\
0.0550000 & -0.2100000 &  1.1550000 \\
3.2550000 & -8.6100000 &  6.3550000 \\
\end{array}\right]
```

More text processing

See the Wikipedia article on "How Perl saved the Human Genome Project" — merging of huge databases from different labs.

Modules

There are a bajillion modules at CPAN (`cpan.perl.org`). You might check for pre-existing modules before inventing something from scratch.

There are interfaces for complex arithmetic, numerical linear algebra (e.g. LAPACK/BLAS), database interfaces, etc. etc.

References

- *Programming Perl* ("The Camel"), O'Reilly.
- `www.perl.org`
- `cpan.perl.org`