Perl and Inline Octave

(or IPC with an interactive application)

Andy Adler

School of Information Technology and Engineering
University of Ottawa

Rating:

Viewer discretion advised

Warning, this talk contains

- Conspicuous use of windows OS
- Occasional insulting comments about Perl and/or Perl people

Background

- I use Perl to manage files, and Octave to crunch numbers.
- Recently, I worked on a project that generated enormous data files, which needed to be processed and then analysed - a perfect task for my two favourite languages.
- Since I'd just heard a mighty cool talk on Inline (YAPC::NA 2001), it seemed clear to me that I needed to write Inline::Octave.

Sample Problem

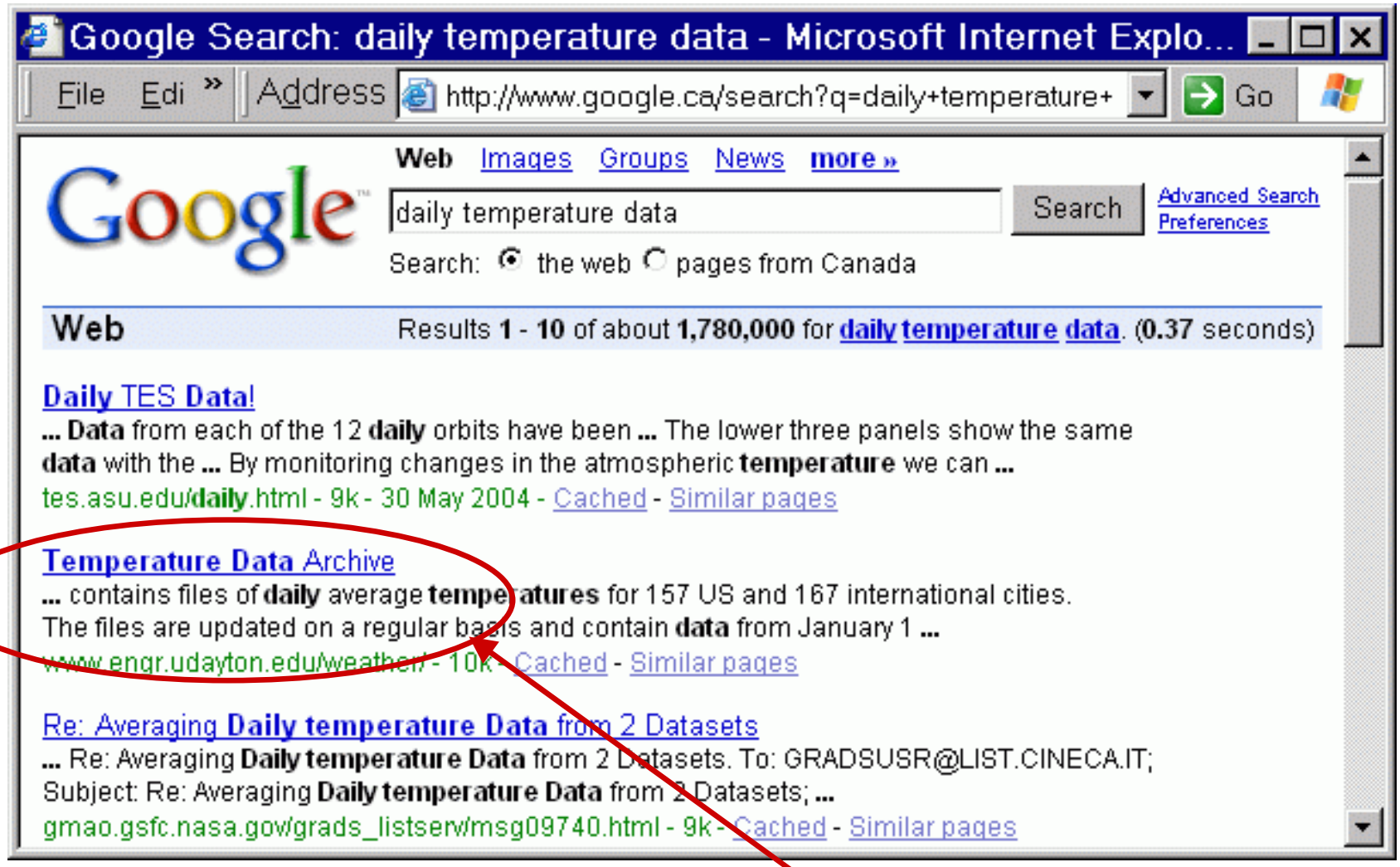
- Is the temperature rising?

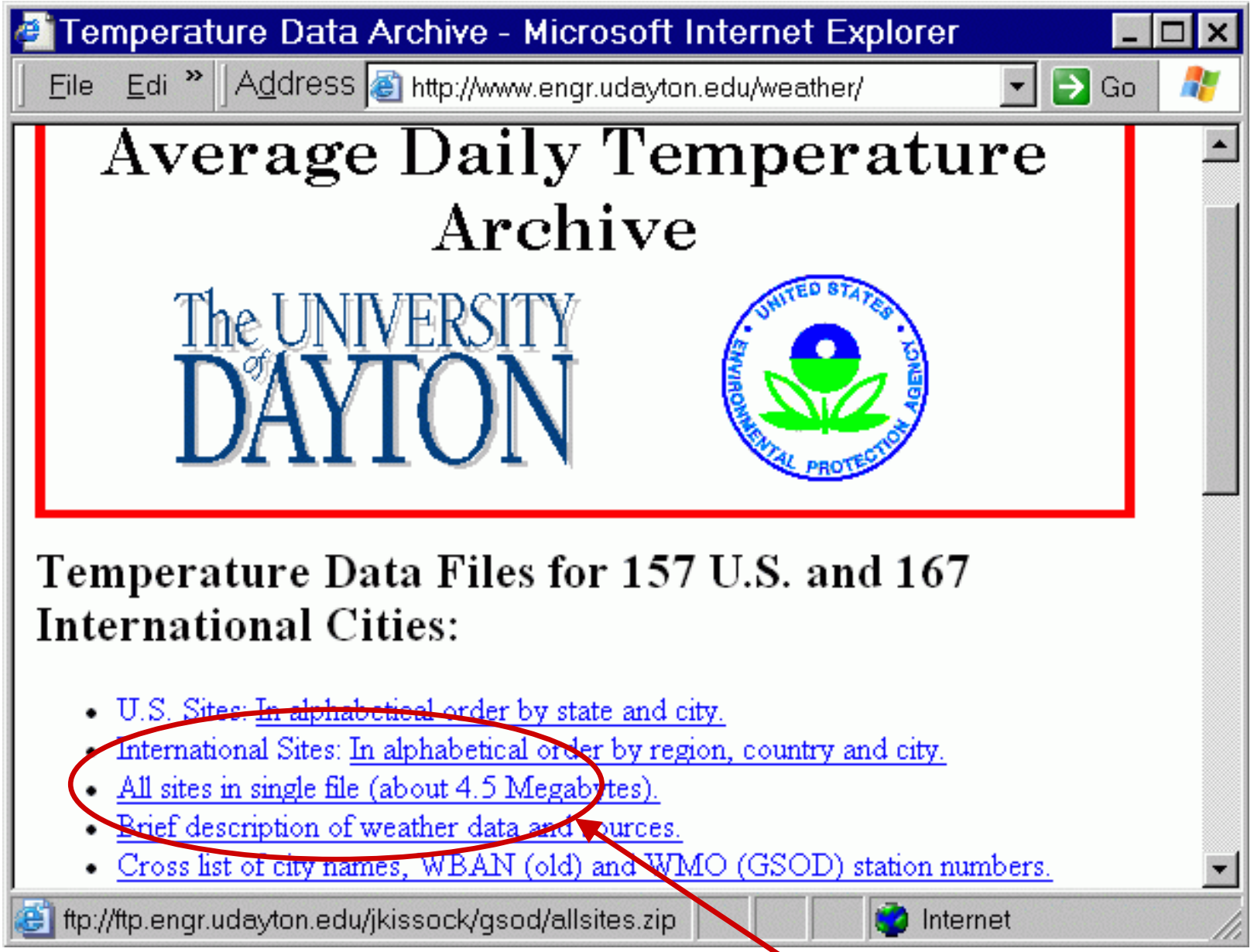
- Let's suppose we've decided that we don't trust those pundits, and we'd like to calculate for ourselves whether the earth is getting warmer.

Sample Problem

- However, for some crazy reason, we do trust random stuff published on the internet

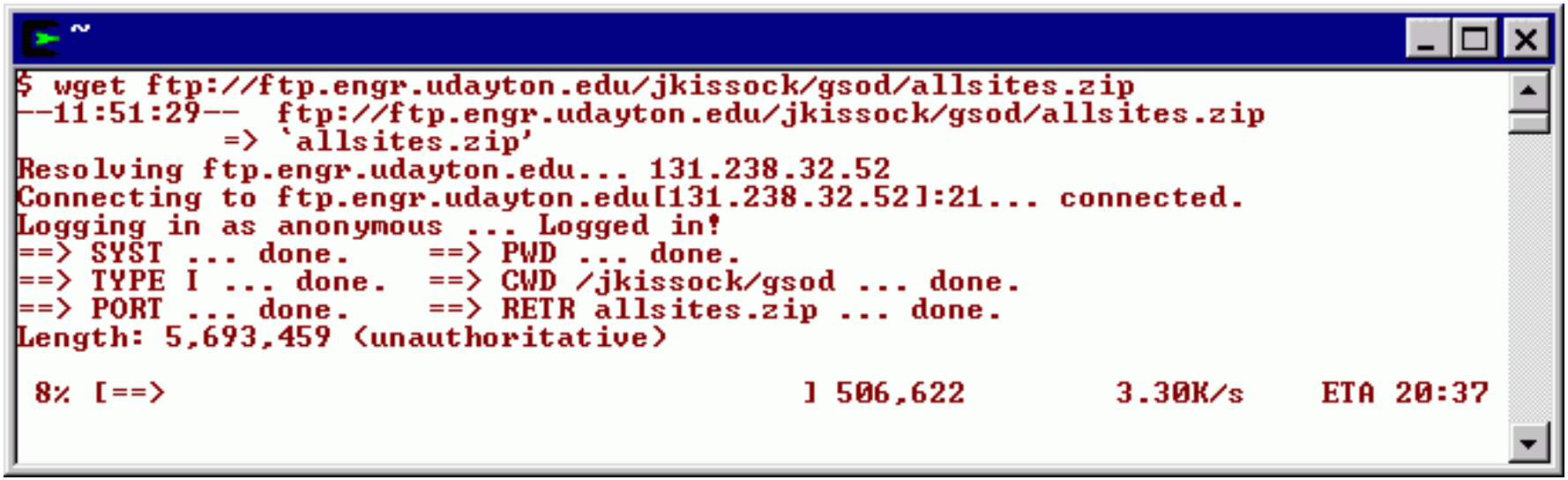
- Type "*daily temperature data*" into google


Google Search: daily temperature data - Microsoft Internet Explo...
File Edi
Address
http://www.google.ca/search?q=daily+temperature+
Go
Web Images Groups News more »
Google
daily temperature data
Search
Advanced Search
Preferences
Search: the web pages from Canada
Web
Results 1 - 10 of about 1,780,000 for daily temperature data. (0.37 seconds)
Daily TES Data!
... Data from each of the 12 daily orbits have been ... The lower three panels show the same data with the ... By monitoring changes in the atmospheric temperature we can ...
tes.asu.edu/daily.html - 9k - 30 May 2004 - Cached - Similar pages
Temperature Data Archive
... contains files of daily average temperatures for 157 US and 167 international cities. The files are updated on a regular basis and contain data from January 1 ...
www.engr.udayton.edu/weather/ - 10k - Cached - Similar pages
Re: Averaging Daily temperature Data from 2 Datasets
... Re: Averaging Daily temperature Data from 2 Datasets. To: GRADSUSR@LIST.CINECA.IT; Subject: Re: Averaging Daily temperature Data from 2 Datasets; ...
gmao.gsfc.nasa.gov/grads_listserv/msg09740.html - 9k - Cached - Similar pages
This looks interesting

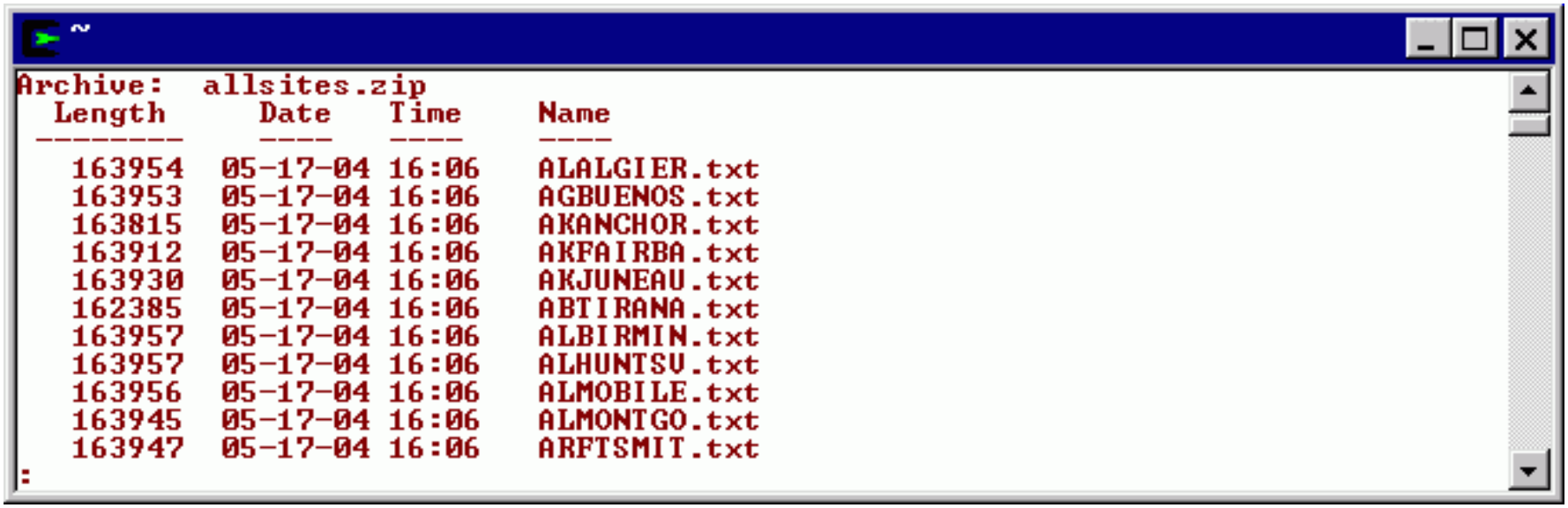


This looks interesting

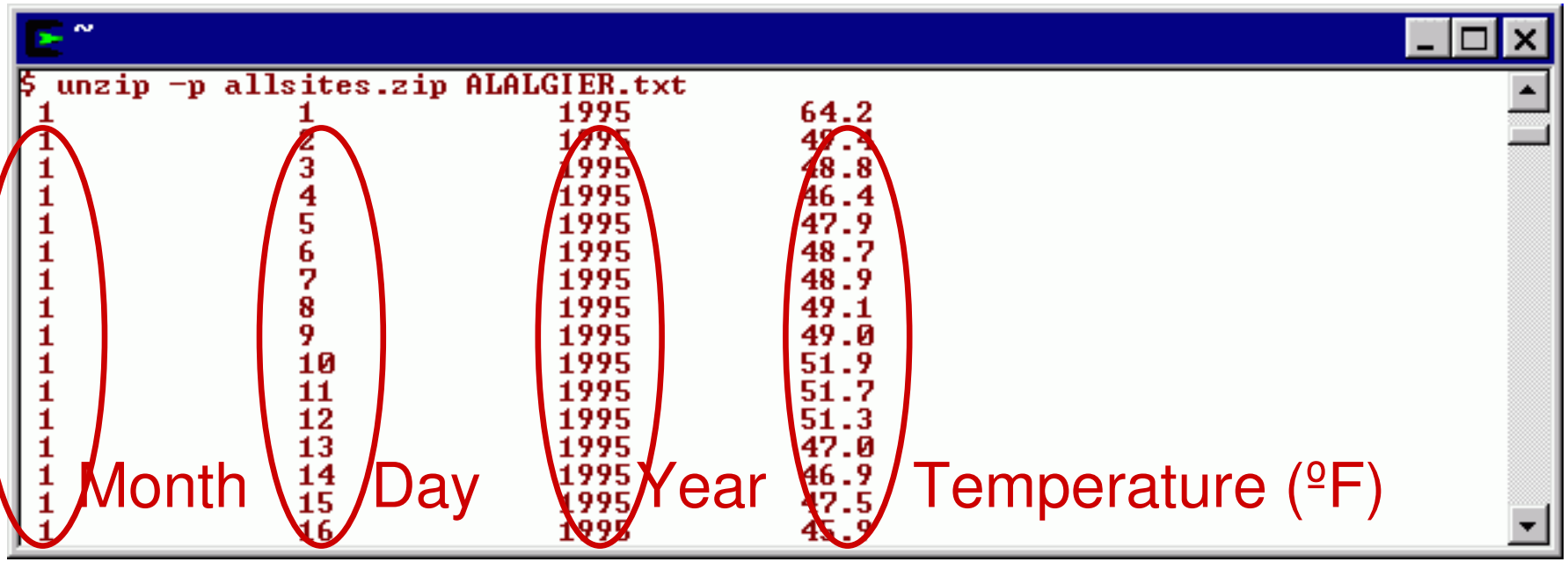
Step 1: download file



Step 2: look at contents



Step 3: file contents



Step 4: parse in Perl

```
if (/^\s* (\d+) \s+ (\d+) \s+ (\d+) \s+ ([\-\d\.]+)/x) {
   next if $4 == -99;
   push @time, timegm( 0, 0, 12, $2, $1-1, $3);
   push @temp, ($4-32)/1.8;
}
```

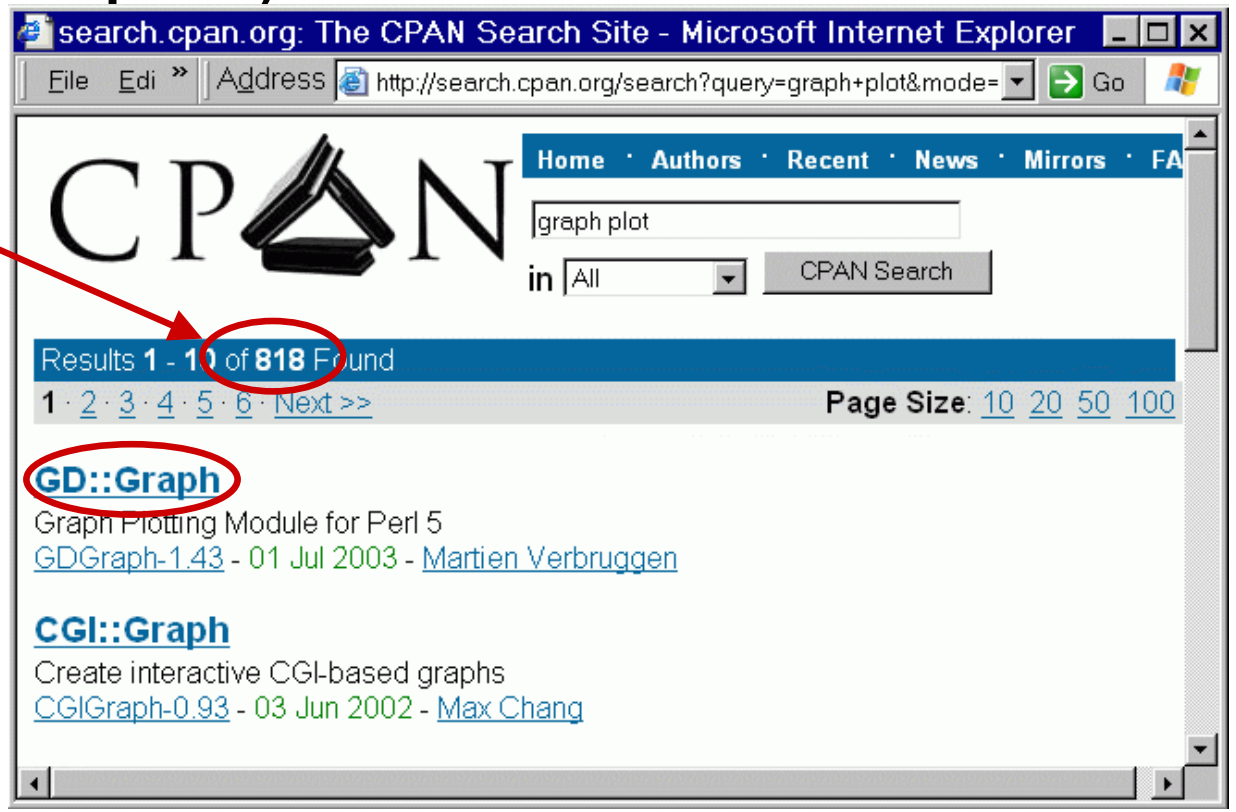
Code for no data

Convert to ºC

What does data look like?

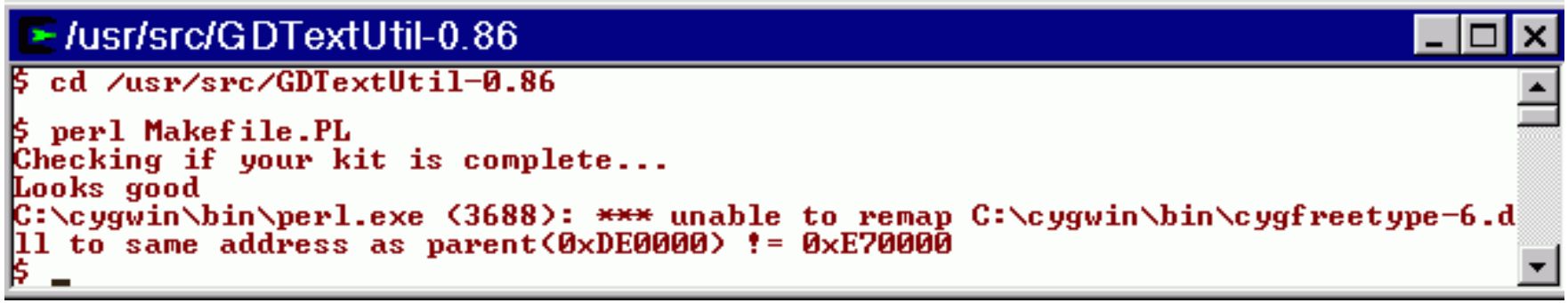
Many ways to plot data

- Load into spreadsheet (OOcalc, Excel)
- Octave (uses gnuplot)
- Perl
  - Lots of ways
  - Try GD::Graph



Plot data with Perl?

- CPAN: Get GD::Graph
- Requires: GD
- Requires: libgd, libjpeg, libpng, libX11 …
- Requires: GD::Text



- This road leads to “DLL hell”.
  Time to try a detour

Plot data via Octave via Gnuplot

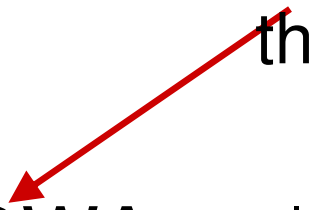
Octave plot syntax

```
Xdata= [x1,x2,x3];
Ydata= [y1,y2,y3];
plot(Xdata, Ydata);
```

Usage:

```
unzip –p allsites.zip CNOTTOWA.txt | \
    perl showplot.pl | \
    octave -q
```

Note misspelling:
I wonder how reliable these data are?



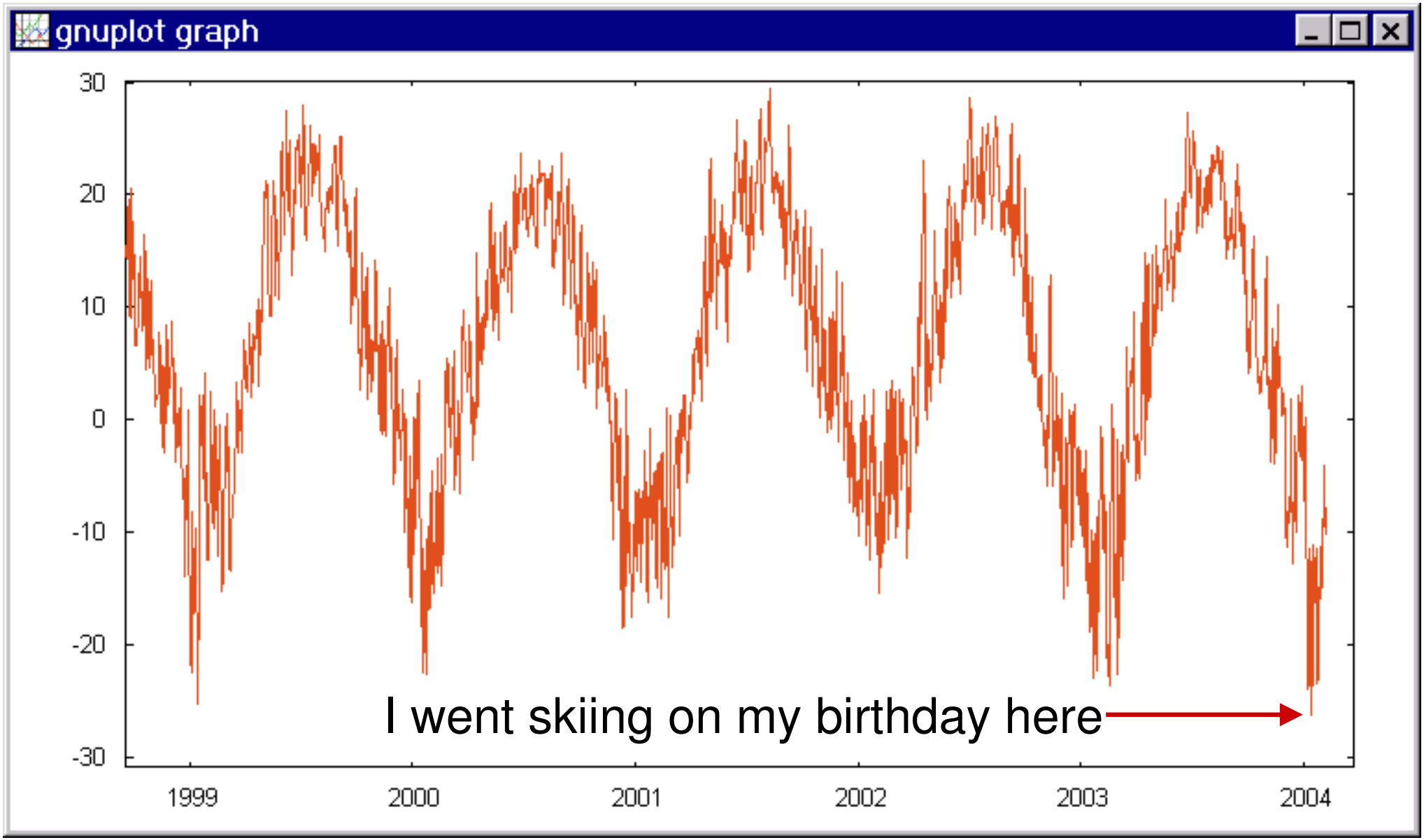
Plot code

```
#!perl -w
use Time::Local;
my $y2k= timegm( 0, 0, 0, 1, 0, 2000);
while (<>) {
      if (/^\s+ (\d+) \s+ (\d+) \s+ (\d+) \s+ ([\-\d\.]+)/x) {
          next if $4 == -99;
          my $sec_y2k= timegm( 0, 0, 12, $2, $1-1, $3) - $y2k;
          push @time, 2000 + $sec_y2k / 365.2422/24/60/60;
          push @temp, ($4-32)/1.8;
      }
}
print    " Xdata=[",join(",",@time),"];\n",
         " Ydata=[",join(",",@temp),"];\n",
         " plot(Xdata, Ydata);\n";
```

Convert from epoch to beginning 2000

Exact days per year

Temp (ºC): Ottawa for 1999-2004



How to calculate trend

- Problem:

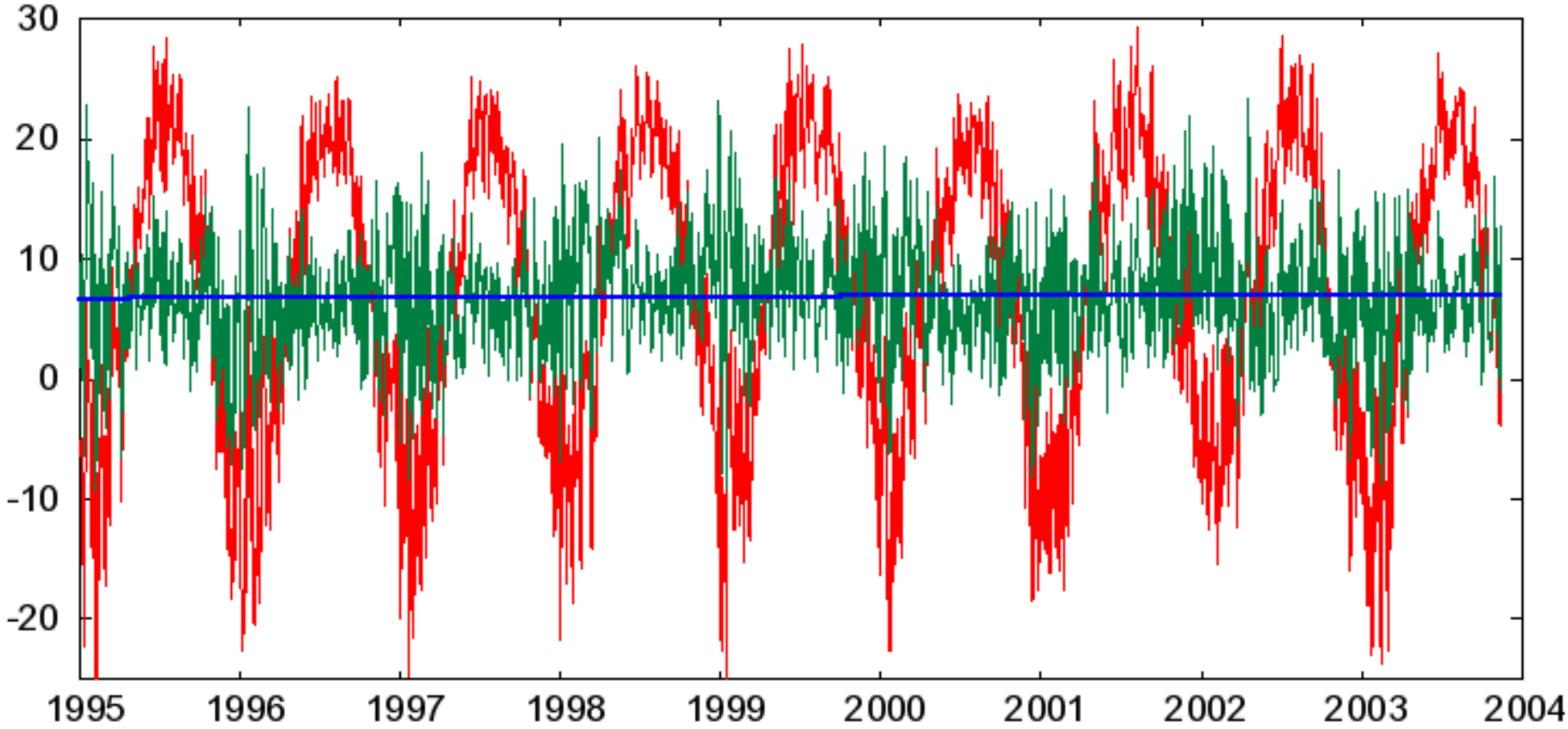
  Intra-year variations are much larger than year to year variations

- Approach:

  extract and remove signal component in phase with the year. Remainder is the trend

- Notes:
  - This can be done in perl – but I think it's easier to use a mathematical language
  - This is not the *correct* way to calculate temperature trends. Please refer to the scientific literature

Analysis: remove year harmonics



Red: Temp (ºC) vs. time (years) in Ottawa.

Green: Components in phase with the year removed

Blue: Best fit line

Perl code: process input / output

```
use Time::Local; my ($city, @time, @temp, %rates);
open F, "unzip -c $ARGV[0] |" or die …
while (<F>) {
        if (/^\s+ (\d+) \s+ (\d+) \s+ (\d+) \s+ ([\-\d\.]+)/x) {
                push @time, … push @temp, …
        }
        if (/^\s+ inflating: \s+ (\w+) \.txt/x) {
                process(\@time, \@temp) if $city;
                $city= $1; @time= (); @temp= ();
        }
}       process(\@time, \@temp); #last one
close F;
printf "Rate: %1.4f±%1.4f (°C/year) for %d cities\n",
        calc_stats()->as_list;
```

"unzip –c" outputs all files with "inflating" between each

process and *calc_stats* do data analysis

Octave code: *calc_stats*

Perl

```
printf "Rate: %1.4f±%1.4f (°C/year) for %d cities\n",
        calc_stats()->as_list;
```

Octave

```
function stat_data= calc_stats()
        global sites;
        n_cities= length(cities);
        stat_data= [   mean(cities),
                       std(cities)/sqrt(n_cities),
                       n_cities];
```

Octave code: *process*

Perl

```
process(\@time, \@temp);
```

Octave

```
function TperYr= process( time, temp );
  global sites=[]; static site_no=1;

  time_step= [0;diff(time)/2] + [diff(time)/2;0];
  harmonics= 2*pi*(1:3) / ( 365.2422*24*60*60 );
  year_osc= [ sin(time * harmonics), cos(time * harmonics) ] ...
                .* (time_step * ones(1,2*length(harmonics)));
  component= (temp' * year_osc) ./ sumsq( year_osc );
  temp_clean= temp - year_osc * component'; # harmonics

  fit = polyfit( time, temp_clean, 1); # fit to line
  cities( site_no++ ) = fit(1) * ( 365.2422*24*60*60 ); # deg/year
```

Linking Octave code to perl

```
use Time::Local;
open F, "unzip -c $ARGV[0] |" or die …
while (<F>) {
        # stuff
}       process(\@time, \@temp);
close F;
printf "Rate:%1.4f±%1.4f (°C/year) for %d cities\n",
        calc_stats()->as_list;

use Inline Octave => q{
        # Octave code (previous slide)
};
```

Results

Yes, the average temperature is increasing

but not everywhere

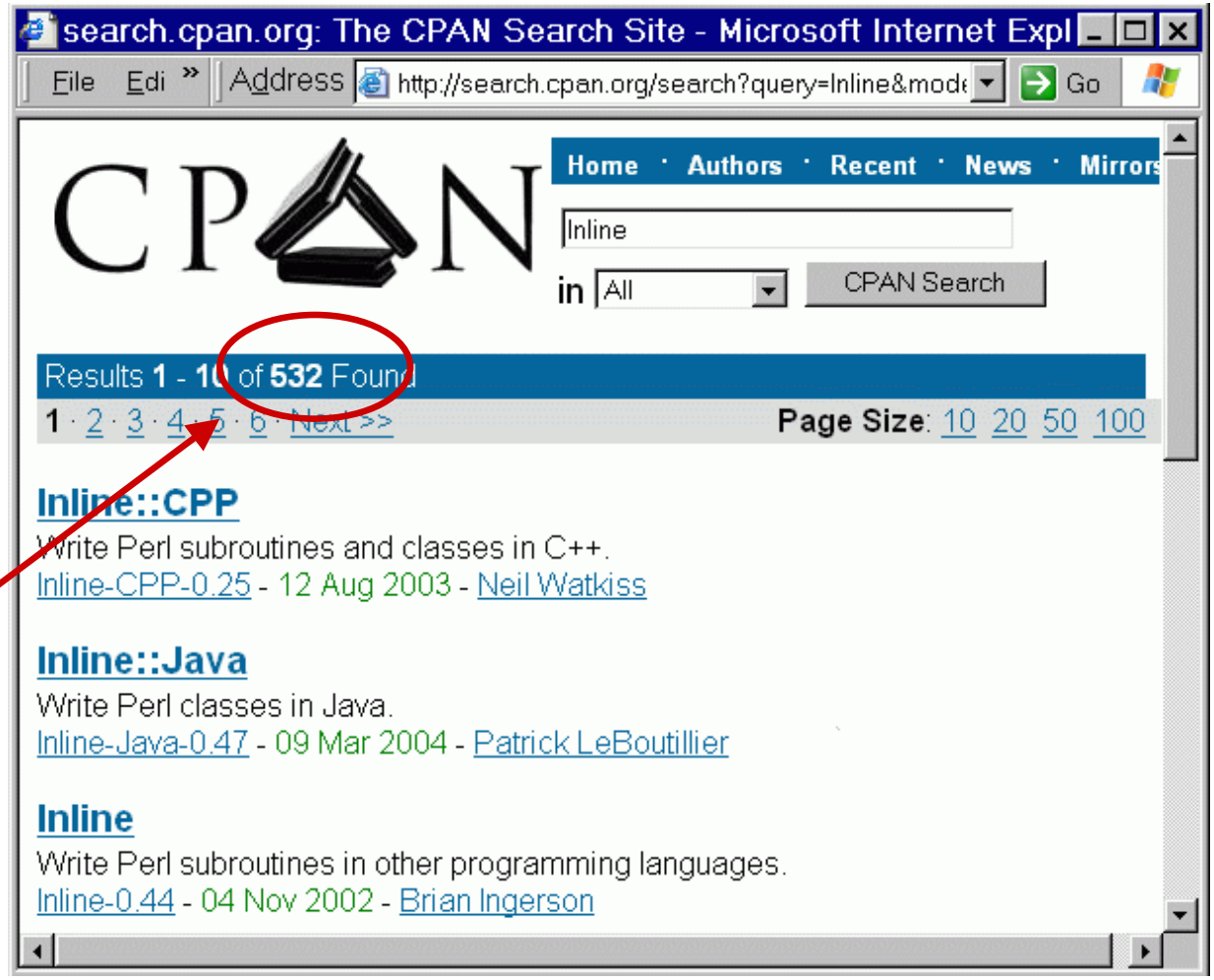
Time: (using P4 2.4Ghz machine)

Cygwin / WinXP	192.5 sec
Linux (Knoppix 3.3)	75.5 sec

A closer view of the glue

Inline

- Infrastructure to write perl in other languages
- Many Inline modules



Inline::C example

```
use Inline C;
print "9 + 16 = ", add(9, 16), "\n";
__END__
__C__
int add(int x, int y) {
  return x + y;
}
```

- C code is extracted and compiled to a dynamic library (or shared object)
- At run-time, perl is linked to dll (or so)

Inline Java example

```
use Inline Java => q[   class Pod_alu {
                            public Pod_alu() {  }
                            public int add(int i, int j){ return i + j; }
                        }  ];
my $alu = new Pod_alu() ;
print $alu->add(9, 16) . "\n“;
```

- Java code is extracted and compiled
- Two possibilities
  - Run code in JVM and interface via sockets
  - Link perl to JVM dll and make calls into it

Octave

- Octave is an interpreted language
  - Syntax is like Matlab
- Specializes in mathematical functions
- Why not use perl (ie. PDL)
  - Efficiency
  - Lots of code Octave/Matlab available
  - There's more than one way to do it

Warning:

The following slide makes an unfair jab at Perl saints.

Viewer discretion is advised

Math languages take correctness seriously

For example, in the (excellent) Perl Cookbook (Christiansen & Torkington):

> These lines do not catch the IEEE notations of "Infinity" and "NaN", but unless you are worried that the IEEE committee members will stop by your workplace and beat you over the head with copies of the relevant standards, you can probably forget about these strange numbers

Serious math people just can't make jokes about things like that

Interfacing with an interpreter

- An interpreter may be controlled by linking to its *stdin*, *stdout,* and *stderr.*
- Perl module: IPC::Open3
- Documentation says:
  - If you try to read from the child's stdout writer and their stderr writer, you'll have problems with blocking, ...".
- I did try, and I did have problems.

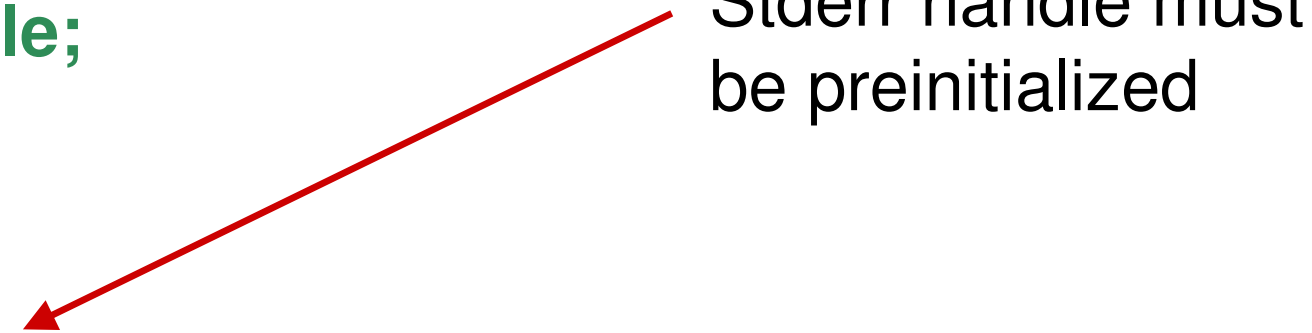
Example.pm

- Usage:
  - *Example::interpret*: send code to interpreter, and capture *stdout* and *stderr*

```
$ perl -MExample -e'print Example::interpret("1/2")'
ans = 0.50000

$ perl -MExample -e'print Example::interpret("1/0")‘
warning: division by zero (in octave code) at -e line 1
ans = Inf
```

Example.pm: *setup*




```
package Example;
use strict;
use Carp;
use IO::Handle;
my $Oerr= new IO::Handle;
use IPC::Open3;
open3( my $Oin, my $Oout, $Oerr, "octave -qH");
setpriority 0,0, (getpriority 0,0)+4; #lower priority slightly
use IO::Select;
my $select = IO::Select->new($Oerr, $Oout);
```

Example.pm: *interpret*

```
my $marker= "-9ABa8l_8Onq,zU9-"; # random string
my $marker_len= length($marker)+1;

sub interpret {
    my $cmd= shift;
    print $Oin "\n\n$cmd\ndisp('$marker');fflush(stdout);\n";
    my $input;
    while ( 1 ) {
        sysread $Oout, (my $line), 16384;
        $input.= $line;
        last if substr( $input, -$marker_len, -1) eq $marker;
    }
    return substr($input , 0 , -$marker_len );
}
```

Make interpreter give back marker when finished

Example.pm: *handling stderr*

- Concept
  - *stdout* is an arbitrary stream of data
  - *stderr* will consist of bursts of error data
- Implementation
  - When we detect stderr data, switch to *process_errors* until finished

Example.pm: *error handler*

```
sub process_errors {
        my $select= IO::Select->new( $Oerr );
        my $input;
        while ( $select->can_read(0.1) ) {
                sysread $Oerr, (my $line), 1024;
                last unless $line;
                $input.= $line;
        }
        croak "$input (in octave code)" if $input =~ /error:/;
        carp "$input (in octave code)" if $input;
}
```

Timeout for stderr data stream

Simple test to detect warnings and errors

Example.pm: *handling stderr*

```
sub interpret {
        my $cmd= shift;
        my $marker= "-9Ahv87uhBa8l_8Onq,zU9-"; # random string
        my $marker_len= length($marker)+1;
        print $Oin "\n\n$cmd\ndisp('$marker');fflush(stdout);\n";
        my $input;
        while ( 1 ) {
                for my $fh ( $select->can_read() ) {
                        if ($fh eq $Oerr) {
                                process_errors();
                        } else {
                                sysread $fh, (my $line), 16384;
                                $input.= $line;
                        }
                }
                last if substr( $input, -$marker_len, -1) eq $marker;
        }
        return substr($input , 0 , -$marker_len );
}
```

Example.pm

- Usage:
  - *Example::interpret*: send code to interpreter, and capture *stdout* and *stderr*

```
$ perl -MExample -e'print Example::interpret("1/2")'
ans = 0.50000

$ perl -MExample -e'print Example::interpret("1/0")‘
warning: division by zero (in octave code) at -e line 1
ans = Inf
```

Conclusion

- Inline is a great way to glue different languages together
- Inline::Octave is one option to do mathematical work in Perl
- Controlling an interpreted language is tricky. However perl allows this with IPC::Open3