Doing mathematics with Tcl

Arjen Markus[1]
WL | Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

Abstract

A handheld calculator is a great way to do all kinds of ad hoc calculations. You do not write a C or Fortran program with hard-coded numbers to add them for balancing your books. In a similar way doing mathematics - for instance solve some geometrical problem - requires a handheld calculator suitable for that type of calculations.

The essential ingredients are an interactive program that interprets dedicated commands, and a few conventions to make its use more or less uniform.

This paper describes the application of Tkcon and such a set of conventions to provide the interface. It illustrates the concepts with three different mathematical tools: complex numbers, ordinary differential equations and (statistical) optimisation.

Introduction

Mathematical tools such as complex numbers or differential equations are used to solve specific problems, The nature of these problems may vary widely and thus any flexible computer program that is used to solve them requires an appropriate tool-specific *language* to describe the problem at hand and to specify the steps by which to arrive at a solution. A general program to solve differential equations numerically will consist of one or more methods for solving the equations and of some kind of interpreter or translator for the language in which the equations are expressed.

Scripting languages are ideal for at least the task of interpreting the input. For example, the equation for a dampened oscillator:

```
m x'' + r x' + k x = 0
```

Initial conditions:

```
x = A, x' = B at t = 0
```

(the prime indicates a time-derivative)

can easily be translated into Tcl, be it ad hoc:

---

[1] E-mail address: arjen.markus@wldelft.nl

- Variables x and y (=x') are the state variables of the problem (m, r, k, A and B are fixed parameters)
- Then: *expr {$y}* evaluates to the derivative of x
- And: *expr {-($r * $y +$k * $x) / $m) }* evaluates to the derivative of y
- At the start of the calculation x and y are set to $A and $B respectively:
  set x $A ; set y $B ; set t 0.0
- With a suitable loop we can perform the integration over time, say using the first-order method by Euler:

```
set x $A
set y $B
set t 0.0
while { $t < $tend } {
       set dx [expr {$y}]
       set dy [expr {-($r * $y +$k * $x) / $m) }]
       set x [expr {$x+$dt*$dx}]
       set y [expr {$x'+$dt*$dy]
       set t [expr {$t+$dt}]
       puts "$t $x $y"
}
```

As a scripting language does not need separate compilation and linking steps in order to get an executable program, you can rely on the language's runtime environment to interpret the expressions - you do not need to program this yourself. Thus, with some added mechanisms, these expressions can be readily typed in and the equations solved.

The framework

The "workbench" that is the subject of this paper provides a simple framework for implementing specific mathematical tools, such as the ordinary differential equations. It relies on standard components like *Tkcon*, a set of conventions and a handful of general commands to create uniformity among the tools[2] (after all, it is meant to be used interactively).

Tkcon is used as a command window: you can type in Tcl commands, view the results and use the command history mechanism to retrieve and modify previous commands. It offers a large number of other facilities as well, but these are the ones which make it suitable as a "calculator".

Now, let us examine the general commands:

- [overview] gives an overview of available commands, variables and modules:

```
> overview vars
> overview commands
> overview modules
```

- [type] returns the type of a variable:

```
> set v [complex 1.0 -1.0]
> type v
COMPLEX
```

- [print] formats the contents of a variable in a form suitable for that type of variable:

```
> set v [complex 1.0 -1.0]
```

[2] For lack of a better term, the implementation of a mathematical tool is called a *module* in the workbench.

```
> print v
1.0-1.0i
```

- [display] presents the contents of the variable in a graphical way
- [help] presents a short description of a command (its purpose and its arguments)

Individual modules define their own specific commands, for instance the *complex numbers* module:

- [complex] constructs a complex number that is then stored in an ordinary Tcl variable:

  ```
  > set v [complex 1.0 -1.0]
  ```

  Because it uses the conventions of the workbench, the variable contains information about its type:

  ```
  > puts $v
  {COMPLEX 1.0 -1.0}
  ```

- [add], [sub], [mult], [div] define arithmetic operations on these complex numbers:

  ```
  > set w [add $v [complex 0.1 2.5]]
  ```

- Operations for extracting the real or imaginary part and such also exist, [real], [imag], [conjugate], [arg], [rad], with evident meanings.
- [phifunc] is a command to create complex numbers from a given angle (phi, a reserved name):

  ```
  > set f [phifunc {1.0+cos($phi)}]    ;# $f defines a cardioid
  ```

  As a result a function $f is created that takes one argument, the angle in radians, calculates the radius from the given function and then returns the complex number with that angle and radius:

  ```
  > set w [$f 1.3]
  > print w
  0.339054+1.22131i
  ```

- Similarly [zfunc] is a command to create a new complex function that evaluates some expression in z:

  ```
  > set f [zfunc {mult $z $z}]    ;# $f squares its argument
  ```

The implementation of these commands relies on a simple convention: the contents of the variable is either a list where the first element is the name of of the type or it is a new string of form "typename##uniquenumber". The latter form makes it easier to create new functions such as in [phifunc] above.

To illustrate:

```
proc complex {{real 0.0} {imag 0.0}} {
   list COMPLEX $real $imag
}

proc add {val1 val2} {
   set r1 [real $val1]
   set i1 [imag $val1]
   set r2 [real $val2]
   set i2 [imag $val2]
   complex [expr {$r1+$r2}] [expr {$i1+$i2}]
```

```
}

proc phifunc {expression} {
   set name [::Workbench::uniqueID "COMPLEX-PHI-FUNCTION"]
   interp alias {} $name {} [namespace current]::phifunc_impl $expression
   return $name
}

proc phifunc_impl {expression phi} {
   set rad [expr $expression]
   set x   [expr {$rad*cos($phi)}]
   set y   [expr {$rad*sin($phi)}]
   complex $x $y
}
```

(The implementation of commands like [phifunc] and [zfunc] uses the versatile [interp alias] command as a macro expander for Tcl - you can define implicit arguments with it.)

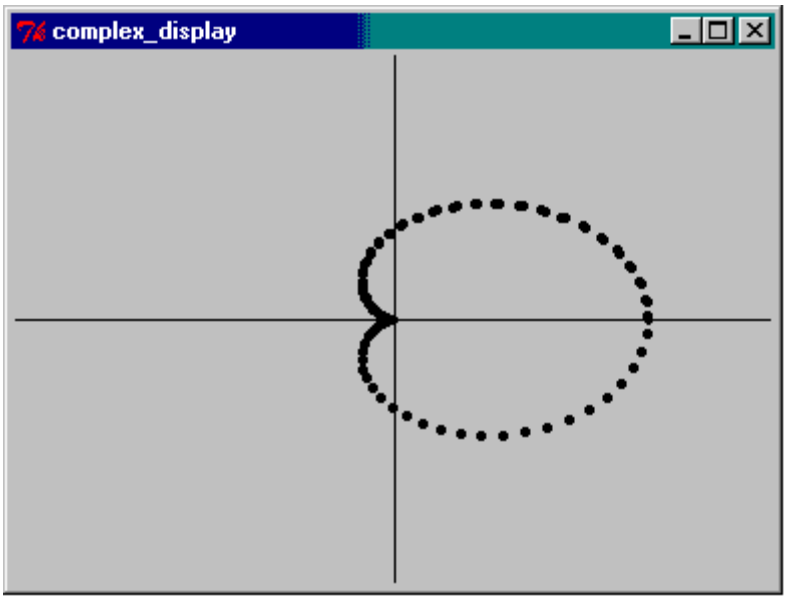
General commands like [print] and [display] rely on this convention to delegate the actually printing or graphical representation to type-specific commands:

```
proc printComplex {name} {
   upvar $name var
   set vtype [uplevel type $name]
   switch -- $vtype {
   "COMPLEX" {
      return "[format "%g%+gi" [real $var] [imag $var]]"
   }
   "COMPLEX-PHI-FUNCTION" {
      return "Polar function: radius = [lindex [interp alias {} $var] 1]"
   }
   "COMPLEX-Z-FUNCTION" {
      return "Mapping of z using procedure [lindex [interp alias {} $var] 1]"
   }
   default {
      return "$var"
   }
   }
}

#
# Register the type-specific commands

::Workbench::moduleType "COMPLEX" {
   print   ::ComplexNumbers::printComplex
   display ::ComplexNumbers::displayComplex
   edit    ::ComplexNumbers::editComplex
}
```

Of course, all the facilities that Tcl offers, like for-loops are still available:

```
for { set phi 0.0 } { $phi < 10.0 } { set phi [expr {$phi+0.1}] } {
   set z [$f $phi]
   display z
}
```

produces the picture below



More on complex numbers

In both complex and real analysis the ability to combine functions is of paramount importance - it allows one to build ever more complicated functions and still be able to analyse their properties. For instance:

$$f(x) = \sin(x),\ g(x) = x^2$$
$$(f \circ g)(x) = f(g(x)) = \sin(x^2)$$
$$(g \circ f)(x) = g(f(x)) = \sin^2(x)$$

This can be done in the framework of Tcl as well:

```
set f [zfunc {sin $z}]
set g [zfunc {mult $z $z}]
set h [compose $f $g]
set k [compose $g $f]
```

The [compose] command is generally applicable, as long as functions are of compatible types. The full implementation of [compose] checks for this compatibility by checking the list of registered type combinations, but the essence of the command is simple (again we use [interp alias]):

```
proc compose {func_a func_b} {
   upvar $func_a fa
   upvar $func_b fb

   set name [uniqueID "COMPOSITE"]
   interp alias {} $name {} [namespace current]::ComposeImpl \
      $compose_accept($type_a,$type_b) $fa $fb
   return $name
}

proc ComposeImpl {return_type func_a func_b args} {
   return [$func_a [eval $func_b $args]]
}
```

The reason for the compability check is that the result of the first function must be taken as input to the next, but other than that, there is virtually no limitation.

Ordinary differential equations

The second module that I wish to discuss in some detail is one that solves (systems) of ordinary differential equations - in a much more elegant and general fashion than the simple example from the introduction.

Again look at the equation for a dampened oscillator:

```
m x'' + r x' + k x = 0
```

Initial conditions:

```
x = A, x' = B at t = 0
```

Or, when written as a system of first-order equations:

```
dx/dt   = y
m dy/dt = -r y - k x
```

Initial conditions:

```
x = A, y = B at t = 0
```

To describe this system, we need the following concepts:

- A *state variable*, the dependent variable for which a differential equation muist be solved and that needs to be given an initial value.
- A *parameter*, a fixed value within the equations that can only vary between calculations.

For manipulating these two types of variables several commands have been defined:

- [statevar] defines a new state variable (and en passant the initial value and the differential equation):

```
> set A 1.0 ; set B 0.0

> statevar x $A {$y}
> statevar y $B {-$r*$y-$k*$x}
```

- [parameter] defines a parameter with its value. Such parameters are substituted into the equations when solving the system.

```
> parameter r 0.1
```

- [solve] solves the defined system of equations:

```
> solve $tstart $tstop
```

  After the completion of the command, [display] can be used to draw the solution.
- Other commands are available for convenience:
  - [initial] to redefine the initial value of a state variable
  - [timestep], [outputstep] control the time step of the calculation and the frequency of the output
  - [method] selects which solution method to use

The effect of these commands is that they create several Tcl variables (one for each piece of information) and register the name in a list:

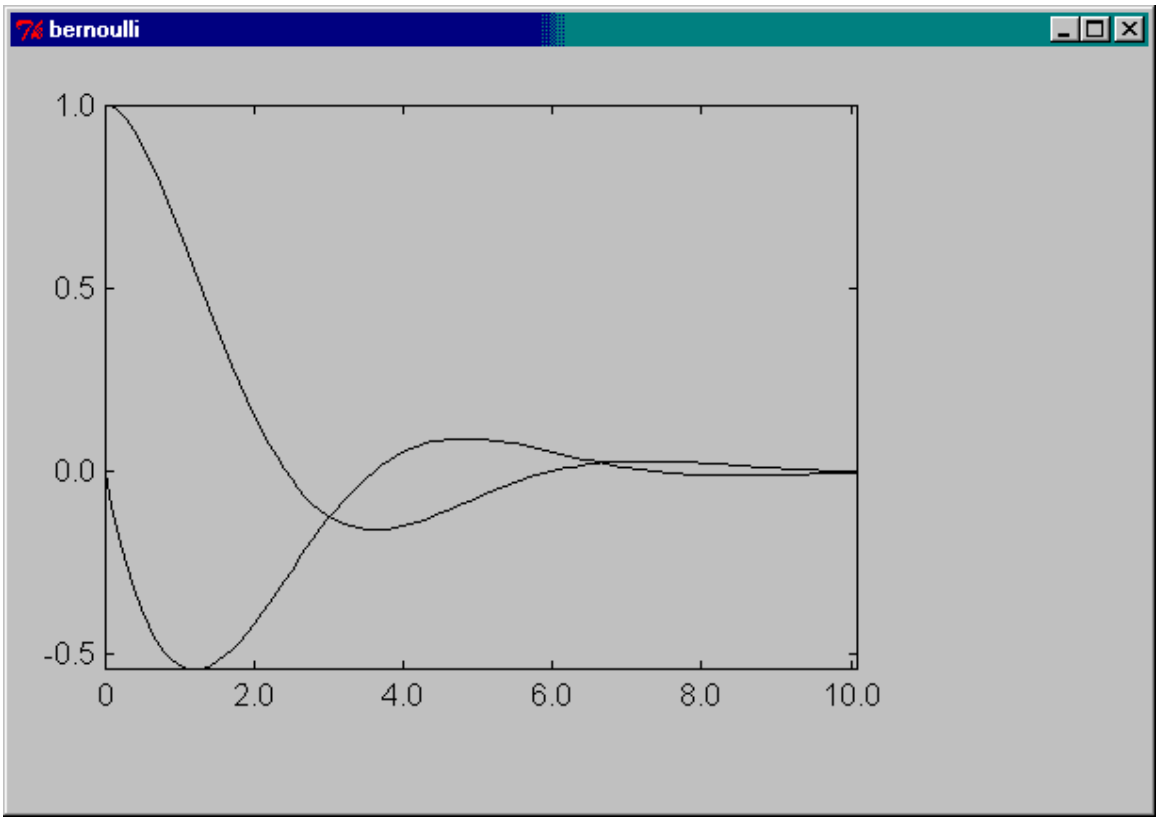
```
proc statevar { name initval deriv } {
   variable list_state_vars

   variable Data::$name   $initval
   variable Init::$name   $initval
   variable Deriv::$name  $deriv
   variable Result::$name {}

   if { [lsearch $list_state_vars $name] == -1 } {
      lappend list_state_vars $name
   }
   uplevel [list set $name [list STATEVARIABLE $name]]
}
```

The registered Tcl variables are then used in the command [solve] to construct the actual system of equations. For the numerical solution the workbench relies on the Tcllib module "::math::calculus":

- [solve] initialises all the variables to their proper value. Then it constructs lists of the variables for use in a private procedure.
- This private procedure evaluates all the equations as prescribed by the ::math::calculus module.
- In a loop in [solve] the new values are determined and stored for later use.

Thus, the above equations lead to the following result:

```
> statevar x 1.0 {$y}
> statevar y 0.0 {-$y-$x}
> timestep 0.1
> solve 0.0 10.0
> display x
> display y
```



The module is named *bernoulli*, after the famous mathematicians who were among the pioneers of this field of mathematics. With *bernoulli* you can easily define systems of ordinary differential equations and solve them.

Optimisation

The third example involves the analysis of results from, say, a computer program with the specific purpose of optimising the parameters controlling the computation. Numerical models of physical and biological phenomena, like weather prediction or ecosystem dynamics, often contain parameters that require calibration. In principle the idea behind calibration is simple: chose a set of values, determine a solution, compare this to the measurements and change the values if the difference is too large. Then rerun the model until you can be satisfied. If the calculations are lengthy or if there are several parameters to choose from, then this seemingly simple procedure takes a lot of effort. Statistical analysis techniques, like *maximum likelihood estimation*, can be used to make the selection of new values easier. This is precisely the purpose of *SPACE*, a program written by Matthias Schönlau (*cf.* Literature) and freely available via the Internet.

The program itself has a command-line interface which is fairly easy to use but which is also unique to the program. Basically it allows the user to read tables of data into memory and perform all kinds of analyses on them, such as:

- *cross-validation:* Can a data point be predicted from the other data points? This gives insight in the quality of the data set. If the prediction errors are large, then the fitted model will not be accurate. On the other hand, very small errors may indicate that the data points are too close together or fail to capture the variation of the function one has sampled.

- *design points:* Indicate where to look for improvements in the result. From the estimates of these so-called design points, the user can decide whether it is worthwhile to continue the calibration and with what set of parameter values or to stop if no significant improvement is likely.

Such optimisation is only one technique one can apply to tables of data and the SPACE program is only one program with its own set of commands. Another example is solving linear programs, a class of optimisation problems with constraints. Therefore the module that I developed to work with this particular program consists of three parts:

- a public and general set of procedures for handling the data (both input and results)
- a private set of procedures that interface with the actual program
- a set of "receipes": a user will probably want to be able to extend or modify the analyses. The receipes allow this without having to rewrite the module itself.

Now the public commands include:

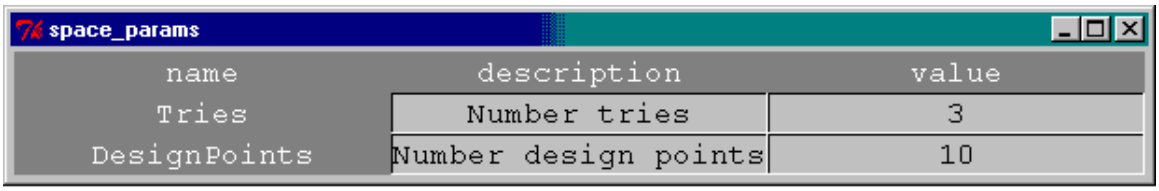
- [dataTable] to define a new table
- [readTable] to read data into the table from a file, and of course [writeTable] to store them into another file
- [editTable] to view the contents of a table and to modify the data if necessary
- [analyseTable] to run a particular analysis on the data
- [displayTable] to provide a simple XY-graph of two selected columns in a table
- [showReceipes] to list the available receipes
- [receipe] to register a particular receipe
- [editParams] to edit the (SPACE-specific) parameters that gouvern the analyses
- [makeRange] to create a table that contains the ranges in which to search for solutions (part of some types of analysis)
- [activeRange] to select a *range table* for subsequent analyses

If you want to interface to a different program (or library) that also handles tabular data, the private procedures will need to be adjusted, but (hopefully) the public procedures remain the same.

To handle the tabular data I use the ::struct::matrix command from Tcllib and the extension Tktable. The combination is quite powerful - it provides a suitable data structure to handle arbitrary two-dimensional tables, and in the philosphy of Tcl, it does not matter whether the table contains strings or numbers (or both). Bringing up the table in a window for editing is simple enough, as shown by the procedure below (*_params_array_* is a Tcl array, whereas *params_matrix* is the command that represents the matrix of analysis parameters)

```
proc editParams { } {
   variable _params_array_

   toplevel .space_params
   table .space_params.edit_table \
      -variable [namespace current]::_params_array_ \
      -cols [params_matrix columns] -rows [params_matrix rows] \
      -titlerows 1 -titlecols 1 -font "Courier 12" \
      -colwidth 20
   pack .space_params.edit_table -fill both
   bind .space_params.edit_table <Return> break

   params_matrix link -transpose [namespace current]::_params_array_
```

```
    tkwait window .space_params.edit_table
    params_matrix unlink [namespace current]::_params_array_
}
```



The most complicated task is doing the actual analysis:

- The procedure [analyseTable] takes as arguments the name of the table to analyse and the name of the receipe.
- It starts by preparing the input to the program (both the data contained in the table and the filled-in receipe)
- Then it runs the program, feeding it the commands from the receipe. It catches the output from the external program and shows this in a logging window. The main reason for this is to show the progress and possible analysis details that are otherwise filtered out.
- Finally the various results (each analysis receipe may provide its own output) are collected and made available as ordinary data tables for further inspection.

This module is the most extensive of the three presented here, mainly because the interfacing with the external program requires quite some coding. Focus was rather on proving that such an interface could be realised than making it a full-fledged tool.

Concluding remarks

With a shell like *Tkcon* and the power and flexibility of Tcl it is possible to build a "workbench" that supports al kinds of specialised tools. The examples given here are only a small set of what can be achieved with the aid of a few building blocks (see the table below) and a few conventions.

Of course it is no match for widely used commercial packages like MATLAB, but this workbench has much in common with these packages, namely a set of mathematical tools that can be extended at will and a working environment that allows typing in the commands as well using source files when the amount of code is too large.

Module	*Extension*
Complex numbers	Tkcon (Jeff Hobbs)
Ordinary differential equations (bernoulli)	Tkcon emu_graph (Steve Cassidy) ::math::calculus (Arjen Markus)
Optimisation (SPACE)	Tkcon emu_graph Tktable (Jeff Hobbs) ::struct::matrix (in Tcllib, Andreas Kupries) SPACE (Matthias Schönlau)

Literature

M. Schönlau

SPACE, Statistical Process Analysis of Computer Experiments
http://www.schonlau.net