[pageheader "Package: Calculus"]

[synopsis \

{package require Tcl 8.2

package require math::calculus 0.5

::math::calculus::integral begin end nosteps func

::math::calculus::integralExpr begin end nosteps expression

::math::calculus::integral2D xinterval yinterval func

::math::calculus::integral3D xinterval yinterval zinterval func

::math::calculus::eulerStep t tstep xvec func

::math::calculus::heunStep t tstep xvec func

::math::calculus::rungeKuttaStep tstep xvec func

::math::calculus::boundaryValueSecondOrder coeff\_func force\_func leftbnd rightbnd nostep}]

::math::calculus::newtonRaphson func deriv initval

::math::calculus::newtonRaphsonParameters maxiter tolerance

[section "Introduction"]

The package Calculus implements several simple mathematical algorithms,

such as the integration of a function over an interval and the numerical

integration of a system of ordinary differential equations.

[par]

It is fully implemented in Tcl. No particular attention has been paid to

the accuracy of the calculations. Instead, well-known algorithms have

been used in a straightforward manner.

[par]

This document describes the procedures and explains their usage.

[section "Version and copyright"]

This document describes [italic ::math::calculus], version 0.5, may 2002.

[par]

Usage of Calculus is free, as long as you acknowledge the

author, Arjen Markus (e-mail: arjen.markus@wldelft.nl).

[par]

There is no guarantee nor claim that the results are accurate.

[section "Procedures"]

The Calculus package defines the following public procedures:

[ulist]

[item][italic "integral begin end nosteps func"]

[break]

Determine the integral of the given function using the Simpson

rule. The interval for the integration is [lb]begin,end[rb].

[break]

Other arguments:

[break]

[italic nosteps] - Number of steps in which the interval is divided.

[break]

[italic func] - Function to be integrated. It should take one

single argument.

[par]

[item][italic "integralExpr begin end nosteps expression"]

[break]

Similar to the previous proc, this one determines the integral of

the given [italic expression] using the Simpson rule.

The interval for the integration is [lb]begin,end[rb].

[break]

Other arguments:

[break]

[italic nosteps] - Number of steps in which the interval is divided.

[break]

[italic expression] - Expression to be integrated. It should

use the variable "x" as the only variable (the "integrate")

[par]

[item][italic "integral2D xinterval yinterval func"]

[break]

The [italic integral2D] procedure calculates the integral of

a function of two variables over the rectangle given by the

first two arguments, each a list of three items, the start and

stop interval for the variable and the number of steps.

[break]

The currently implemented integration is simple: the function is

evaluated at the centre of each rectangle and the content of

this block is added to the integral. In future this will be

replaced by a bilinear interpolation.

[break]

The function must take two arguments and return the function

value.

[par]

[item][italic "integral3D xinterval yinterval zinterval func"]

[break]

The [italic integral3D] procedure is the three-dimensional

equivalent of [italic intergral2D]. The function taking three

arguments is integrated over the block in 3D space given by the

intervals.

[par]

[item][italic "eulerStep t tstep xvec func"]

[break]

Set a single step in the numerical integration of a system of

differential equations. The method used is Euler's.

[break]

[italic t] - Value of the independent variable (typically time)

at the beginning of the step.

[break]

[italic tstep] - Step size for the independent variable.

[break]

[italic xvec] - List (vector) of dependent values

[break]

[italic func] - Function of t and the dependent values, returning

a list of the derivatives of the dependent values. (The lengths of

xvec and the return value of "func" must match).

[par]

[item][italic "heunStep t tstep xvec func"]

[break]

Set a single step in the numerical integration of a system of

differential equations. The method used is Heun's.

[break]

[italic t] - Value of the independent variable (typically time)

at the beginning of the step.

[break]

[italic tstep] - Step size for the independent variable.

[break]

[italic xvec] - List (vector) of dependent values

[break]

[italic func] - Function of t and the dependent values, returning

a list of the derivatives of the dependent values. (The lengths of

xvec and the return value of "func" must match).

[par]

[item][italic "rungeKuttaStep tstep xvec func"]

[break]

Set a single step in the numerical integration of a system of

differential equations. The method used is Runge-Kutta 4th

order.

[break]

[italic t] - Value of the independent variable (typically time)

at the beginning of the step.

[break]

[italic tstep] - Step size for the independent variable.

[break]

[italic xvec] - List (vector) of dependent values

[break]

[italic func] - Function of t and the dependent values, returning

a list of the derivatives of the dependent values. (The lengths of

xvec and the return value of "func" must match).

[par]

[item][italic "boundaryValueSecondOrder coeff\_func force\_func leftbnd rightbnd nostep"]

[break]

Solve a second order linear differential equation with boundary

values at two sides. The equation has to be of the form:

[preserve]

d dy d

-- A(x)-- + -- B(x)y + C(x)y = D(x)

dx dx dx

[endpreserve]

Ordinarily, such an equation would be written as:

[preserve]

d2y dy

a(x)--- + b(x)-- + c(x) y = D(x)

dx2 dx

[endpreserve]

The first form is easier to discretise (by integrating over a

finite volume) than the second form. The relation between the two

forms is fairly straightforward:

[preserve]

A(x) = a(x)

B(x) = b(x) - a'(x)

C(x) = c(x) - B'(x) = c(x) - b'(x) + a''(x)

[endpreserve]

Because of the differentiation, however, it is much easier to ask

the user to provide the functions A, B and C directly.

[break]

[italic coeff\_func] - Procedure returning the three coefficients

(A, B, C) of the equation, taking as its one argument the x-coordinate.

[italic force\_func] - Procedure returning the right-hand side

(D) as a function of the x-coordinate.

[italic leftbnd] - A list of two values: the x-coordinate of the

left boundary and the value at that boundary.

[italic rightbnd] - A list of two values: the x-coordinate of the

right boundary and the value at that boundary.

[italic nostep] - Number of steps by which to discretise the

interval.

The procedure returns a list of x-coordinates and the approximated

values of the solution.

[par]

[item][italic "solveTriDiagonal acoeff bcoeff ccoeff dvalue"]

[break]

Solve a system of linear equations Ax = b with A a tridiagonal

matrix. Returns the solution as a list.

[break]

[italic acoeff] - List of values on the lower diagonal

[italic bcoeff] - List of values on the main diagonal

[italic ccoeff] - List of values on the upper diagonal

[italic dvalue] - List of values on the righthand-side

[par]

[item][italic "newtonRaphson func deriv initval"]

[break]

Determine the root of an equation given by [italic "f(x) = 0"],

using the Newton-Raphson method.

[break]

[italic func] - Name of the procedure that calculates the function value

[italic deriv - Name of the procedure that calculates the derivative of the function

[italic initval] - Initial value for the iteration

[par]

[item][italic "newtonRaphsonParameters maxiter tolerance"]

[break]

Set new values for the two parameters that gouvern the Newton-Raphson method.

[break]

[italic maxiter] - Maximum number of iterations

[italic tolerance] - Relative error in the calculation

[par]

[endlist]

[italic Notes:]

[break]

Several of the above procedures take the [italic names] of procedures as

arguments. To avoid problems with the [italic visibility] of these

procedures, the fully-qualified name of these procedures is determined

inside the calculus routines. For the user this has only one

consequence: the named procedure must be visible in the calling

procedure. For instance:

[preserve]

namespace eval ::mySpace {

namespace export calcfunc

proc calcfunc { x } { return $x }

}

#

# Use a fully-qualified name

#

namespace eval ::myCalc {

proc detIntegral { begin end } {

return [lb]integral $begin $end 100 ::mySpace::calcfunc[rb]

}

}

#

# Import the name

#

namespace eval ::myCalc {

namespace import ::mySpace::calcfunc

proc detIntegral { begin end } {

return [lb]integral $begin $end 100 calcfunc[rb]

}

}

[endpreserve]

[par]

Enhancements for the second-order boundary value problem:

[ulist]

[item]Other types of boundary conditions (zero gradient, zero flux)

[item]Other schematisation of the first-order term (now central

differences are used, but upstream differences might be useful too).

[endlist]

[section Examples]

Let us take a few simple examples:

[par]

Integrate x over the interval [lb]0,100[rb] (20 steps):

[preserve]

proc linear\_func { x } { return $x }

puts "Integral: [lb]::math::calculus::Integral 0 100 20 linear\_func[rb]"

[endpreserve]

For simple functions, the alternative could be:

[preserve]

puts "Integral: [lb]::math::calculus::IntegralExpr 0 100 20 {$x}[rb]"

[endpreserve]

Do not forget the braces!

[par]

The differential equation for a dampened oscillator:

[preserve]

x'' + rx' + wx = 0

[endpreserve]

can be split into a system of first-order equations:

[preserve]

x' = y

y' = -ry - wx

[endpreserve]

Then this system can be solved with code like this:

[preserve]

proc dampened\_oscillator { t xvec } {

set x [lb]lindex \$xvec 0[rb]

set x1 [lb]lindex \$xvec 1[rb]

return [lb]list \$x1 [lb]expr {-\$x1-\$x}[rb][rb]

}

set xvec { 1.0 0.0 }

set t 0.0

set tstep 0.1

for { set i 0 } { \$i < 20 } { incr i } {

set result [lb]::math::calculus::eulerStep \$t \$tstep \$xvec dampened\_oscillator[rb]

puts "Result (\$t): \$result"

set t [lb]expr {\$t+\$tstep}[rb]

set xvec \$result

}

[endpreserve]

Suppose we have the boundary value problem:

[preserve]

Dy'' + ky = 0

x = 0: y = 1

x = L: y = 0

[endpreserve]

This boundary value problem could originate from the diffusion of a

decaying substance.

[par]

It can be solved with the following fragment:

[preserve]

proc coeffs { x } { return [lb]list \$::Diff 0.0 \$::decay[rb] }

proc force { x } { return 0.0 }

set Diff 1.0e-2

set decay 0.0001

set length 100.0

set y [lb]::math::calculus::boundaryValueSecondOrder coeffs force {0.0 1.0} \

[lb]list \$length 0.0[rb] 100[rb]

[endpreserve]