```julia
# Load all the required packages
using ODEInterface
using ForwardDiff
@ODEInterface.import_huge
loadODESolvers();

# Define the right-hand function for automatic differentiation
function vdpolAD(x)
    return [x[2],((1-x[1]^2)*x[2]-x[1])*1e6]
end

# Define the system for the solver
function vdpol(t,x,dx)
    dx[:] = vdpolAD(x);
    return nothing
end

# Define the Jacobian function using AD
function getJacobian(t,x,J)
    J[:,:] = ForwardDiff.jacobian(vdpolAD,x);
    return nothing
end

# Flag to check whether plot is to be generated and saved or not
# Also checks if all solvers are successful
printFlag = true;

# Initial conditions
t0 = 0.0; T = [1.0:11.0;]; x0 = [2.0,0.0];

# Get "reference solution"
Tol = 1e-14;
# for Tol < 1e-14 we get the error "TOLERANCES ARE TOO SMALL"
opt = OptionsODE(OPT_EPS=>1.11e-16,OPT_RTOL=>Tol, OPT_ATOL=>Tol,
OPT_RHS_CALLMODE => RHS_CALL_INSITU,
OPT_JACOBIMATRIX => getJacobian);

# Store only the desired component
# Here, only the first component is desired
# The second component is the first derivative of the first component
# due to the fact that it is a second order system.
# Hence error will be taken over the first component only.
x_ref = Array{Float64}(11);

for i=1:11
    (t,x,retcode,stats) = seulex(vdpol,t0, T[i], x0, opt);
    if retcode!=1
        printFlag = false;
        break;
    end
    x_ref[i] = x[1];
end

if printFlag
```

```
# Store the solver names for plotting
solverNames = ["RADAU","RADAU5","SEULEX"];

# Initialize the variables for plots
# f_e = number of function evaluations
f_e = zeros(33,3);
# err = error wrt ref solution over all time steps and components
err = zeros(33,3);

for i =0:32

    # Set the tolerance for current run
    Tol = 10^(-2-i/4);

    # Set solver options
    opt = OptionsODE(OPT_EPS=>1.11e-16,OPT_ATOL=>Tol,OPT_RTOL=>Tol,
    OPT_RHS_CALLMODE => RHS_CALL_INSITU,
    OPT_JACOBIMATRIX=>getJacobian);

    # Restart the solution for each end time
    # to ensure a more accurate solution
    # compared to dense output

    # Solve using RADAU
    x_radau = Array{Float64}(11);
    for j=1:11
        (t,x,retcode,stats) = radau(vdpol,t0, T[j], x0, opt);
        # If solver fails do not continue further
        if retcode != 1
            println("Solver RADAU failed");
            printFlag = false;
            break;
        end
        x_radau[j] = x[1];
        f_e[i+1,1] = stats.vals[13];
    end
    # If solver fails do not continue further
    if !printFlag
        break;
    end
    err[i+1,1] = norm(x_radau-x_ref,Inf);

    # Solve using RADAU5
    x_radau5 = Array{Float64}(11);
    for j=1:11
        (t,x,retcode,stats) = radau5(vdpol,t0, T[j], x0, opt);
        # If solver fails do not continue further
        if retcode != 1
        println("Solver RADAU5 failed");
            printFlag = false;
            break;
        end
        x_radau5[j] = x[1];
        f_e[i+1,2] = stats.vals[13];
```

```
        end
        # If solver fails do not continue further
        if !printFlag
            break;
        end
        err[i+1,2] = norm(x_radau5-x_ref,Inf);

        # Solve using SEULEX
        x_seulex = Array{Float64}(11);
        for j=1:11
            (t,x,retcode,stats) = seulex(vdpol,t0, T[j], x0, opt);
            # If solver fails do not continue further
            if retcode != 1
                println("Solver seulex failed");
                printFlag = false;
                break;
            end
            x_seulex[j] = x[1];
            f_e[i+1,3] = stats.vals[13];
        end
        # If solver fails do not continue further
        if !printFlag
            break;
        end
        # Get the error over all the components and
        err[i+1,3] = norm(x_seulex-x_ref,Inf);
    end

    # Save the plot in PNG format
    # if all the solvers were successful
    if printFlag
        savePlotPNG("vdpolPrecisionTest",f_e,err,solverNames);
    else
        println("Plot cannot be generated due to failure");
    end
else
    println("Plot cannot be generated due to failure of reference solution");
end
```