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

# Define the right-hand function for Automatic Differentiation
function roberAD(x)
    return [-0.04*x[1]+1e4*x[2]*x[3],
        0.04*x[1]-1e4*x[2]*x[3]-3e7*(x[2])^2,
        3*10^7*(x[2])^2]
end

# Define the system for the solver
function rober(t,x,dx)
    dx[1] = -0.04*x[1]+1e4*x[2]*x[3];
    dx[2] = 0.04*x[1]-1e4*x[2]*x[3]-3e7*(x[2])^2;
    dx[3] = 3*10^7*(x[2])^2;
    return nothing
end

# Automatic Differentiation for a more general problem
function getJacobian(t,x,J)
    J[:,:] = ForwardDiff.jacobian(roberAD,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 = 10.^[0.0:11.0;]; x0=[1.0,0.0,0.0];

# Get "reference solution"
# TolMin < 1e-14 gives error "TOLERANCES ARE TOO SMALL"
Tol = 1e-14;
opt = OptionsODE(OPT_RHS_CALLMODE => RHS_CALL_INSITU,
    OPT_RTOL => Tol, OPT_ATOL=>Tol*1e-6,OPT_EPS => 1.11e-16,
    OPT_JACOBIMATRIX => getJacobian);

x_ref = Array{Float64}(12,3)

# Due to a bug in seulex solver for dense output,
# we restart the solution for each required output time
for i=1:12
    (t,x,retcode,stats) = seulex(rober,t0, T[i], x0, opt);
    # If solver fails do not continue further
    if retcode!=1
        printFlag=false;
        break;
    end
    x_ref[i,1] = x[1];
    x_ref[i,2] = x[2];
```

```
    x_ref[i,3] = x[3];
end

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

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

    # Loop over all the tolerances
    for m = 0:32

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

        # Set solver options
        opt = OptionsODE(OPT_EPS=>1.11e-16,OPT_ATOL=>Tol*1e-6,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}(12,3);
        for j=1:12
            (t,x,retcode,stats) = radau(rober,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,1] = x[1];
            x_radau[j,2] = x[2];
            x_radau[j,3] = x[3];
            f_e[m+1,1] = stats.vals[13];
        end
        # If solver fails do not continue further
        if !printFlag
            break;
        end
        err[m+1,1] = norm([norm(x_radau[:,1]-x_ref[:,1],Inf),
            norm(x_radau[:,2]-x_ref[:,2],Inf),
            norm(x_radau[:,3]-x_ref[:,3],Inf)],Inf);

        # Solve using RADAU5
        x_radau5 = Array{Float64}(12,3);
        for j=1:12
```

```
            (t,x,retcode,stats) = radau5(rober,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,1] = x[1];
            x_radau5[j,2] = x[2];
            x_radau5[j,3] = x[3];
            f_e[m+1,2] = stats.vals[13];
        end
        # If solver fails do not continue further
        if !printFlag
            break;
        end
        err[m+1,2] = norm([norm(x_radau5[:,1]-x_ref[:,1],Inf),
            norm(x_radau5[:,2]-x_ref[:,2],Inf),
            norm(x_radau5[:,3]-x_ref[:,3],Inf)],Inf);

        # Solve using SEULEX
        x_seulex = Array{Float64}(12,3);
        for j=1:12
            (t,x,retcode,stats) = seulex(rober,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,1] = x[1];
            x_seulex[j,2] = x[2];
            x_seulex[j,3] = x[3];
            f_e[m+1,3] = stats.vals[13];
        end
        # If solver fails do not continue further
        if !printFlag
            break;
        end
        err[m+1,3] = norm([norm(x_seulex[:,1]-x_ref[:,1],Inf),
            norm(x_seulex[:,2]-x_ref[:,2],Inf),
            norm(x_seulex[:,3]-x_ref[:,3],Inf)],Inf);
    end

    if printFlag
        savePlotPNG("RoberPrecisionTest",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
```