Julia ODEInterface Experiments

Vishal Sontakke

Contents

1 Non-stiff equations **2**
- 1.1 The restricted three body problem (AREN) 2
- 1.2 Movement of hanging rope (ROPE) 3
- 1.3 Code Performance 4
  - 1.3.1 AREN 4
  - 1.3.2 ROPE 5

2 Stiff equations **6**
- 2.1 Van der Pol oscillator (VDPOL) 6
- 2.2 Robertson Chemical Kinetics (ROBER) 7
- 2.3 Code performance 8
  - 2.3.1 VDPOL 8
  - 2.3.2 ROBER 10

3 Appendix of Codes **12**
- 3.1 savePlotPNG 12
- 3.2 ArenPrecisionTest 13
- 3.3 RopePrecisionTest 15
- 3.4 vdpolPrecisionTest 18
- 3.5 RoberPrecisionTest 22

1 Non-stiff equations

The problems chosen for our tests are the following:

1.1 The restricted three body problem (AREN)

We will first look at the restricted three body problem. One considers two bodies of masses $1-\mu$ and μ in circular rotation in a plane and a third body of negligible mass moving around in the same plane. The equations are (see e.g., the classical textbook Szebehely 1967):

$$\begin{aligned}
y_1'' &= y_1 + 2y_2' - \mu' \frac{y_1+\mu}{R_1} - \mu \frac{y_1-\mu'}{R_2}, \\
y_2'' &= y_2 - 2y_1' - \mu' \frac{y_2}{R_1} - \mu \frac{y_2}{R_2}, \\
R_1 &= \left((y_1+\mu)^2 + y_2^2\right)^{3/2}, \quad R_2 = \left((y_1-\mu')^2 + y_2^2\right)^{3/2}, \\
\mu &= 0.012277471, \quad \mu' = 1-\mu.
\end{aligned} \tag{1.1}$$

There exist initial values, for example:

$$\begin{aligned}
y_1(0) &= 0.994, \quad y_1'(0) = 0, \quad y_2(0) = 0, \\
y_2'(0) &= 2.00158510637908252240537862224, \\
t_{end} &= 17.0652165601579625588917206249,
\end{aligned} \tag{1.2}$$

such that the solution is periodic with period t_{end}.

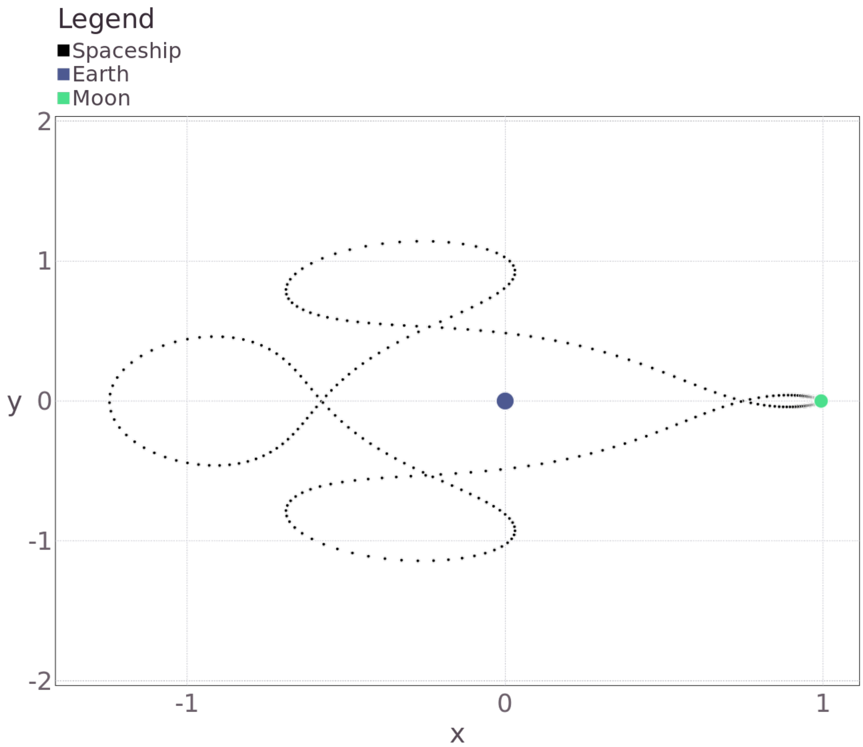

Figure 1.1: An Arenstorf Orbit computed using Dormand & Prince

1.2 Movement of hanging rope (ROPE)

We look at the movement of a hanging rope of length 1 under gravitation and under the influence of a horizontal force

$$F_y(t) = \left(\frac{1}{cosh(4t-2.5)}\right)^4 \tag{1.3}$$

acting at the point $s = 0.75$ as well as a vertical force

$$F_x(t) = 0.4 \tag{1.4}$$

acting at the endpoint $s = 1$.

If this problem is discretized, then Lagrange theory leads to the following equations for the unknown angles θ_k:

$$\begin{aligned}\sum_{k=1}^{n} a_{lk}\ddot{\theta}_k = &-\sum_{k=1}^{n} b_{lk}\dot{\theta}_k^{\,2} - n\left(n+\frac{1}{2}-l\right)\sin\theta_l \\ &- n^2\sin\theta_l \cdot F_x(t) + \begin{cases} n^2\cos\theta_l \cdot F_y(t) & \text{if } l \le 3n/4 \\ 0 & \text{if } l > 3n/4, \end{cases} \qquad l = 1,\dots,n\end{aligned} \tag{1.5}$$

where

$$a_{lk} = g_{lk}\cos(\theta_l - \theta_k), \quad b_{lk} = g_{lk}\sin(\theta_l - \theta_k), \quad g_{lk} = n + \frac{1}{2} - \max(l,k). \tag{1.6}$$

We choose

$$n = 40, \quad \theta_l(0) = \dot{\theta}_l(0) = 0, \quad 0 \le t \le 3.723. \tag{1.7}$$

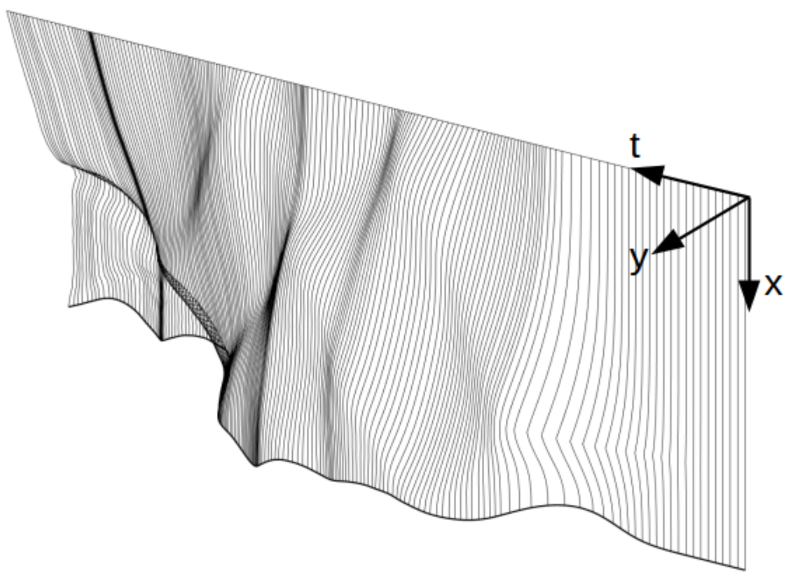

Figure 1.2: Movement of the rope with respect to time

1.3 Code Performance

The three available non-stiff solvers were applied to the above mentioned problems with $Tol = 10^{-3}, Tol = 10^{-3-1/8}, Tol = 10^{-3-2/8}, Tol = 10^{-3-3/8}, \ldots$ up to $Tol = 10^{-14}$, then the numerical result at the output points were compared with a "reference solution" using the infinity norm. The "reference solution" is computed using $Tol = 1.0 \times 10^{-16}$, since quadruple precision is not available on the current hardware (as compared to [1]). The underlying Fortran codes would have to be tweaked to get quadruple precision using software since Julia can emulate Arbitrary Precision floating point numbers using GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library.

1.3.1 AREN

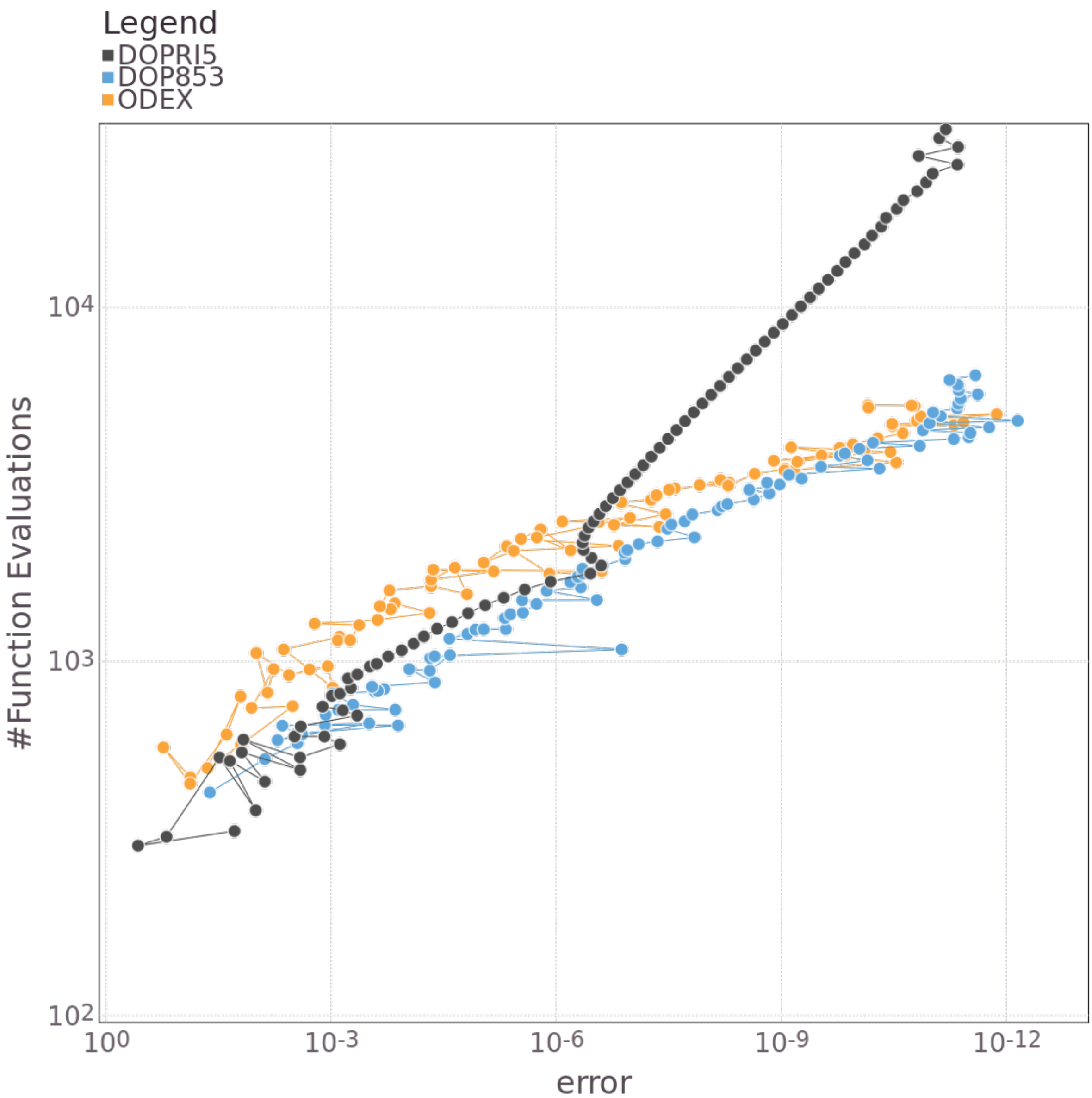

Figure 1.3: Precision vs. Number of function evaluations (as computed on Julia using ODEInterface)

1.3.2 ROPE

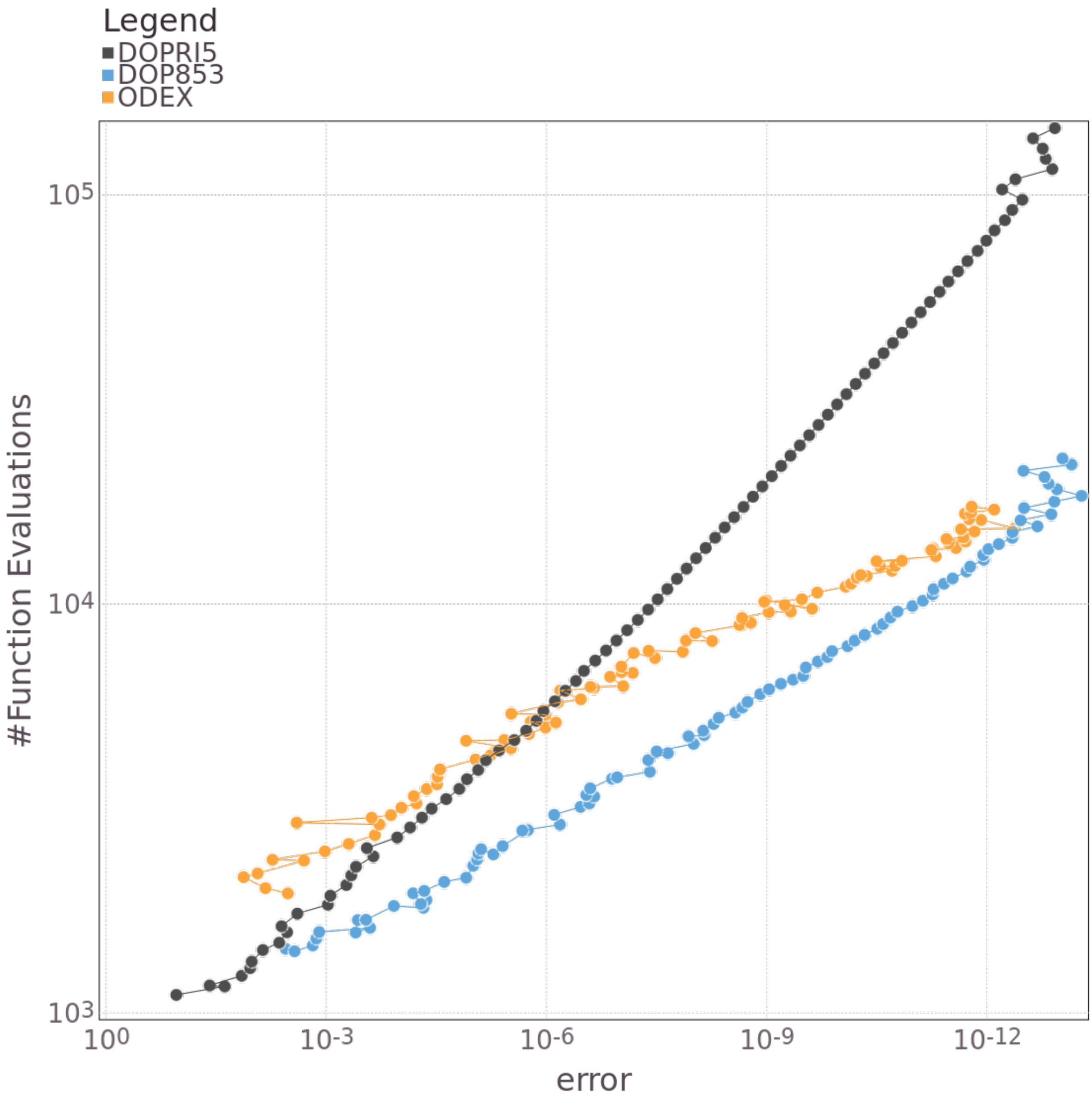

Figure 1.4: Precision vs. Number of function evaluations (as computed on Julia using ODEInterface)

2 Stiff equations

The problems chosen for our tests are the following:

2.1 Van der Pol oscillator (VDPOL)

In dynamics, the Van der Pol oscillator is a non-conservative oscillator with non-linear damping. It evolves in time according to the second-order differential equation:

$$\frac{d^2x}{dt^2} - \mu\left(1 - x^2\right)\frac{dx}{dt} + x = 0, \tag{2.1}$$

where x is the position coordinate-which is a function of time t, and μ is a scalar parameter indicating the non-linearity and the strength of the damping.

The above equation can be converted into a system of first-order differential equations as follows:

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= \left(\left(1 - x_1^2\right)x_2 - x_1\right)/\epsilon, \quad \epsilon = 10^{-6} \end{aligned} \tag{2.2}$$

We perform a rescaling as follows:

$$\begin{aligned} &\tilde{t} = t/\mu, \quad x_1(\tilde{t}) = x(t), \quad x_2(\tilde{t}) = \mu\frac{dx}{dt}(t) \\ \text{and we set} \quad &\frac{1}{\mu^2} = \epsilon \end{aligned} \tag{2.3}$$

This transformation makes the steady state approximation independent of μ.

The initial conditions for this problem are:

$$\begin{aligned} &x_1(0) = 2, \qquad x_2(0) = 0 \\ &t_{out} = 1, 2, 3, 4, \ldots, 11. \end{aligned} \tag{2.4}$$

The times t_{out} will be the points at which the solution will be taken for comparison.

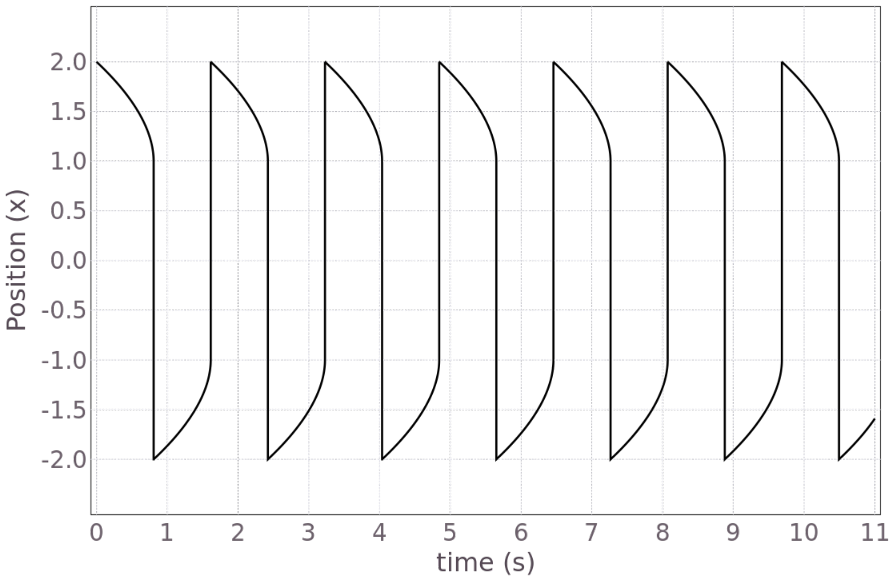

Figure 2.1: Solution of equation (2.1) computed using SEULEX solver

2.2 Robertson Chemical Kinetics (ROBER)

The problem describes the kinetics of an auto-catalytic reaction given by Robertson as follows:

$$\begin{aligned} x_1' &= -0.04x_1 \quad +10^4 x_2 x_3 \\ x_2' &= \quad 0.04x_1 \quad -10^4 x_2 x_3 \quad -3\cdot 10^7 x_2^2 \\ x_3' &= \qquad\qquad\qquad\qquad\qquad 3\cdot 10^7 x_2^2. \end{aligned} \tag{2.5}$$

with initial conditions:

$$x_1(0) = 2, \quad x_2(0) = 0, \quad x_3(0) = 0. \tag{2.6}$$

one of the most prominent examples of the "stiff" literature. It was usually treated on the interval $0 \leq t \leq 40$, until Hindmarsh discovered that many codes fail if t becomes very large (10^{11} say). The reason is that whenever the numerical solution of x_2 accidentally becomes negative, it then tends to $-\infty$ and the run ends by overflow. We have therefore chosen $t_{out} = 1, 10, 10^2, 10^3, \ldots, 10^{11}$.

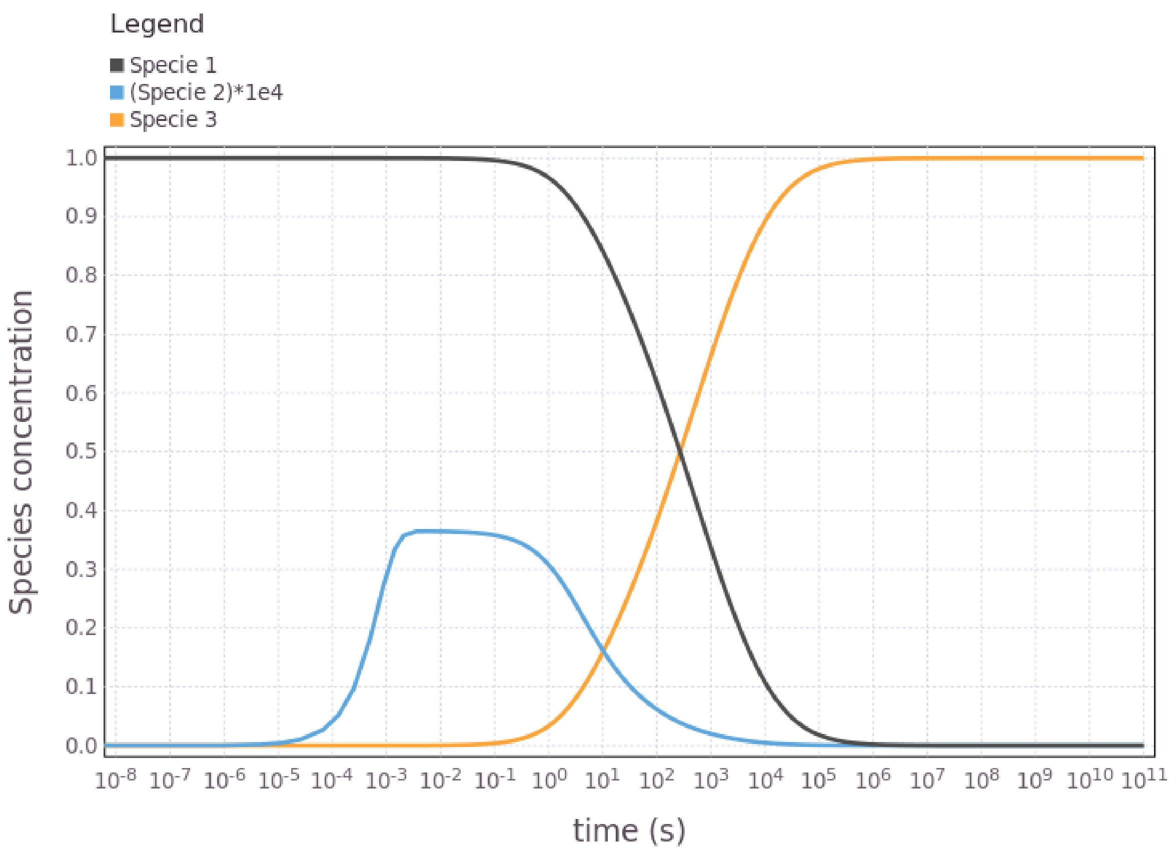

Figure 2.2: Solution of equation (2.5) using SEULEX solver

2.3 Code performance

The three available stiff solvers were applied to the above mentioned problems with $Tol = 10^{-2}, Tol = 10^{-2-1/4}, Tol = 10^{-2-2/4}, Tol = 10^{-2-3/4}, \ldots$ up to $Tol = 10^{-10}$.

We set the relative error tolerance to be $RTol = Tol$ and the absolute error tolerance $ATol = 10^{-6} \cdot Tol$ for the problem ROBER and $ATol = Tol$ for VDPOL. Then the numerical results were compared with a "reference solution" using the infinity norm (the norm is taken over all components and all output points). The "reference solution" is obtained from `http://www.unige.ch/~hairer/testset/testset.html`.

2.3.1 VDPOL

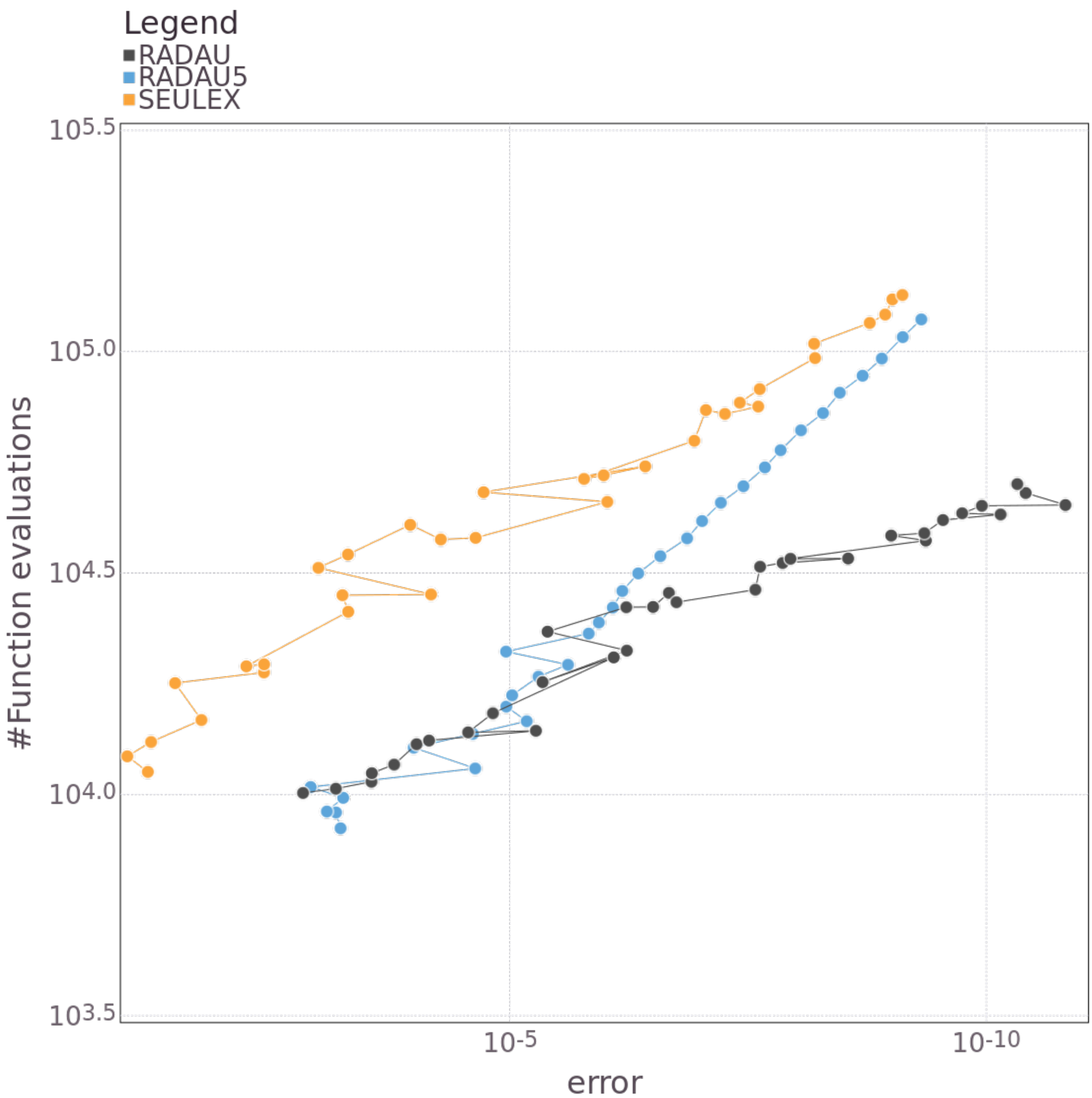

Figure 2.3: Precision vs. Number of function evaluations (as computed on Julia using ODEInterface)

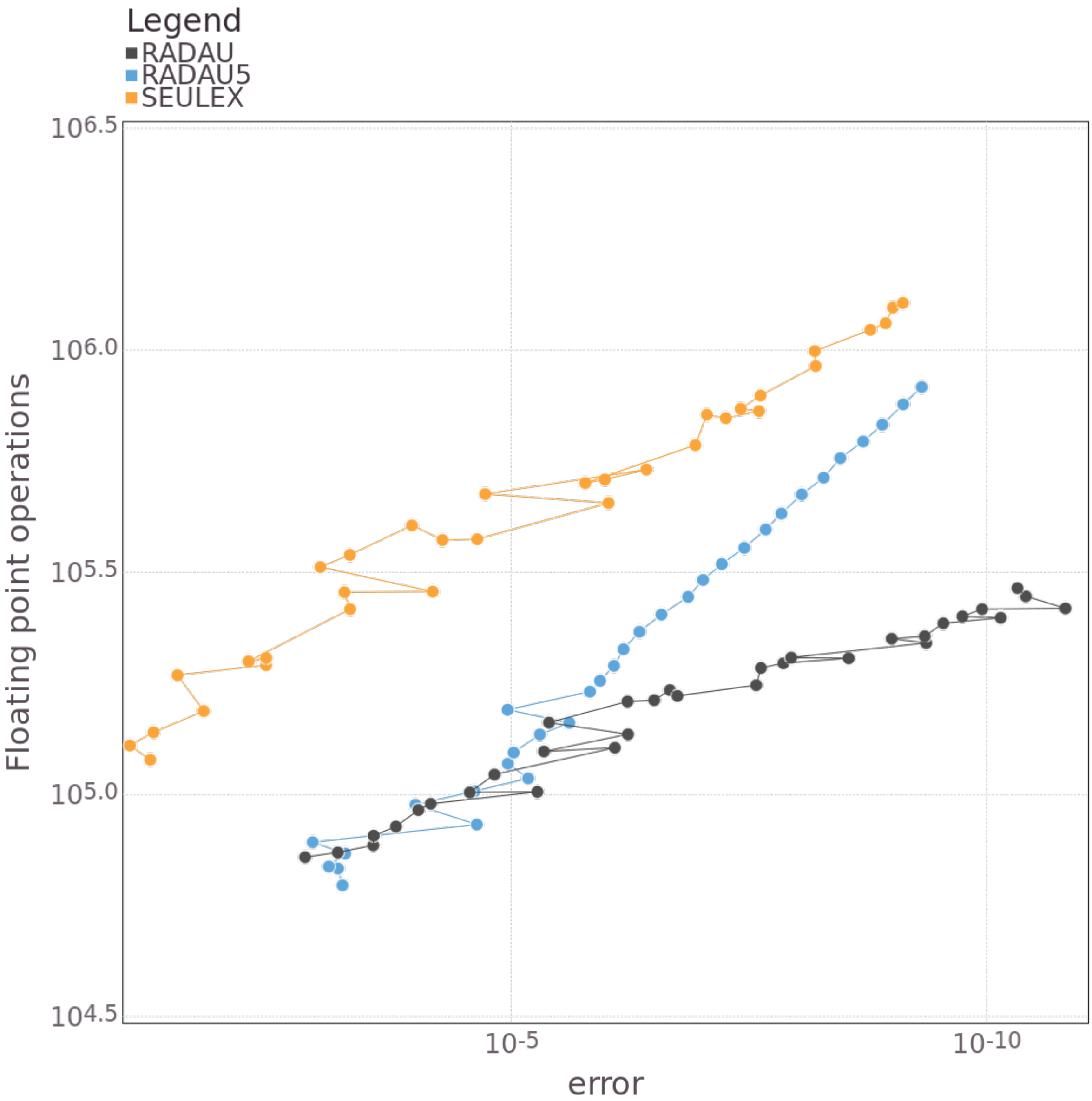

Figure 2.4: Precision vs. Floating point operations (as computed on Julia using ODEInterface)

The following factors were chosen for the various operations:

- Right-hand side function calls = 5 flops
- LU Decomposition = $\lceil \frac{2}{3} \times 8 \rceil = 6$ flops
- Forward or Backward Substitution = 4 flops
- Jacobian computation using Automatic Differentiation = $\lceil 1.5 * 5 \rceil = 8$ flops

2.3.2 ROBER

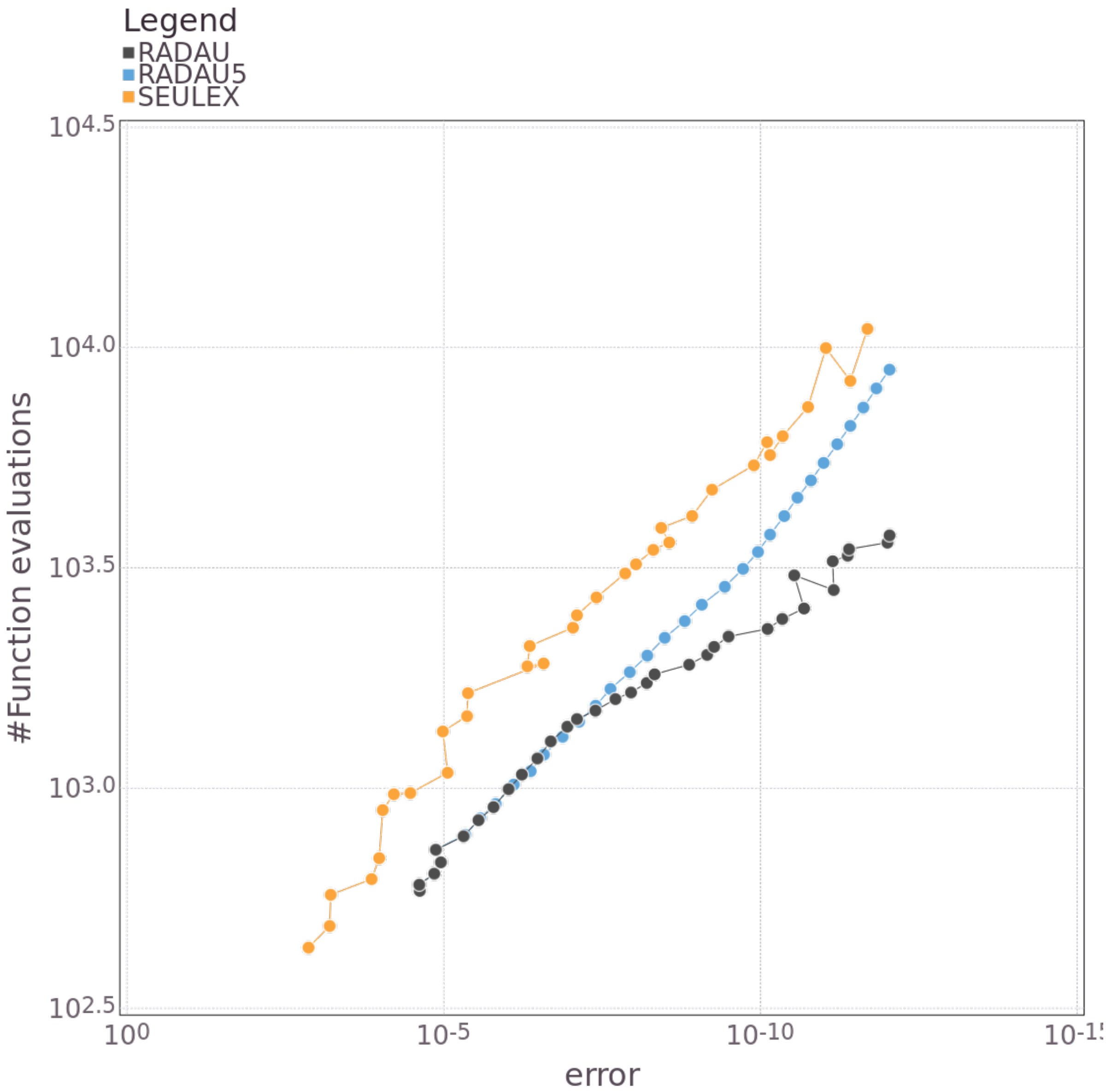

Figure 2.5: Precision vs. Number of function evaluations (as computed on Julia using ODEInterface)

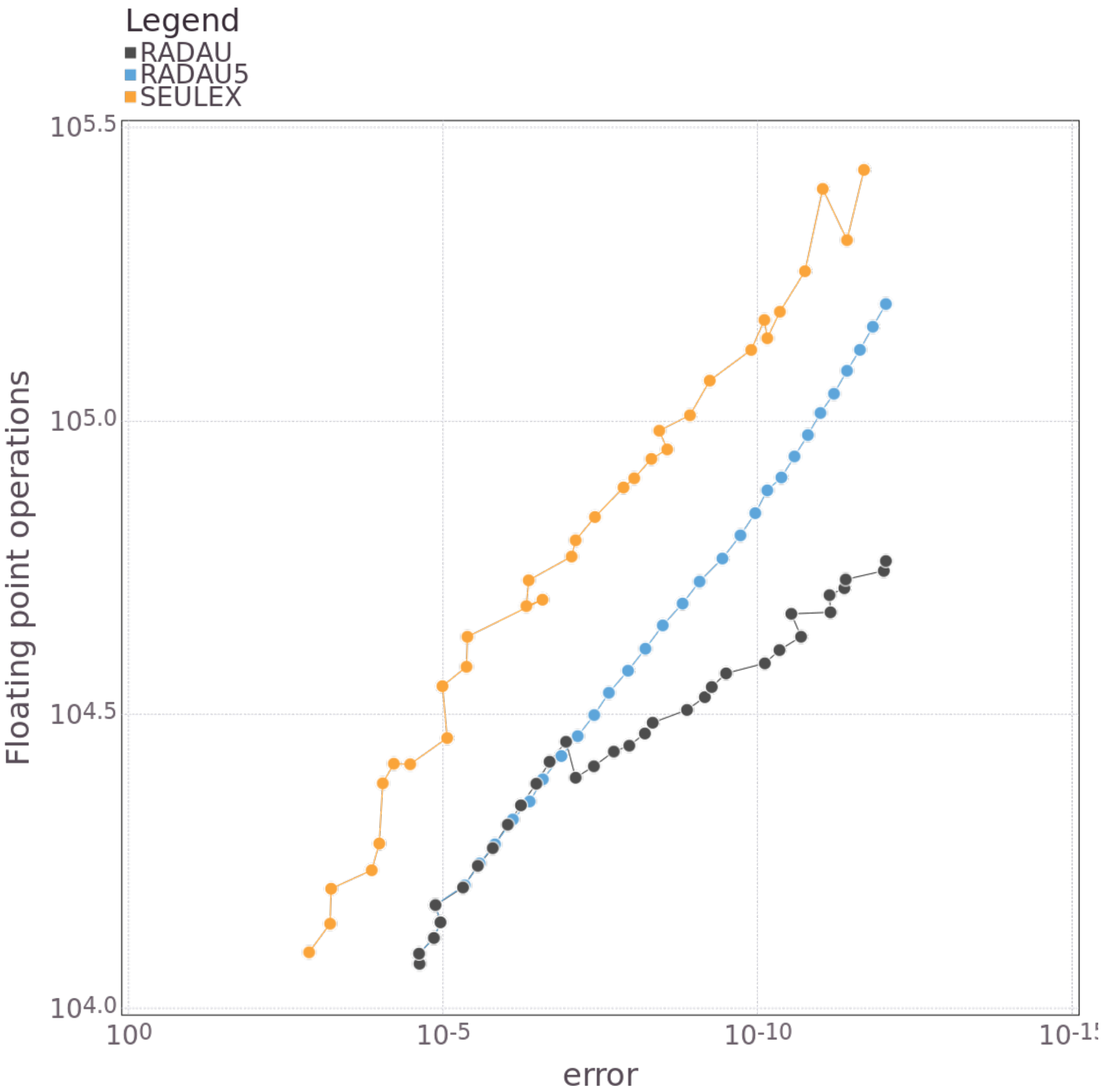

Figure 2.6: Precision vs. Floating point operations (as computed on Julia using ODEInterface)

The following factors were chosen for the various operations:

- Right-hand side function calls = 5 flops
- LU Decomposition = $\lceil \frac{2}{3} \times 8 \rceil = 6$ flops
- Forward or Backward Substitution = 4 flops
- Jacobian computation using Automatic Differentiation = $\lceil 1.5 * 5 \rceil = 8$ flops

3 Appendix of Codes

The codes over here are from March,2016. For updated codes refer https://github.com/sonVishal/ODEInterface.jl/tree/master/myExamples/Codes

3.1 savePlotPNG

```
using Gadfly
using Colors

######################## Function for saving plots #################################
# Input:
# fileName = Name of the file where the plot is to be stored
#            (with or without extension)
# f_e = Array containing function evaluations as columns for each solver
# err = Array containing erros as columns for each solver
# solverNames = Array containing the names of solvers used in respective order
# plotSize = size of the plot to be created
#
# Values have been tuned for a graph similar to the one in
# Solving Ordinary Differential Equations I by
# Hairer, Ernst, Nørsett, Syvert P., Wanner, Gerhard
# page: 252
####################################################################################
function savePlotPNG(fileName,f_e,err,solverNames,
    plotSize=[30cm,30cm])

    numOfLayers = length(solverNames);

    if !contains(fileName,".")
        fileName = string(fileName,".png");
    end

    plotColorsHex = ["#4D4D4D","#5DA5DA","#FAA43A","#60BD68",
        "#F17CB0","#B2912F","#B276B2", "#DECF3F","#F15854"];
    plotColors = [parse(Colorant,c) for c in plotColorsHex];

    majorFontSize = 24pt;
    minorFontSize = 20pt;
    pointSize = 5pt;

    myplot = plot(Scale.x_log10,Scale.y_log10,
        Coord.cartesian(xflip=true),
        Guide.manual_color_key("Legend",solverNames,plotColorsHex[1:numOfLayers]),
        Guide.xlabel("error"),Guide.ylabel("#Function Evaluations"),
        Theme(major_label_font_size=majorFontSize,panel_stroke=colorant"black",
        minor_label_font_size=minorFontSize,key_title_font_size=majorFontSize,
        key_label_font_size=minorFontSize,key_position=:top,key_max_columns=1));

    for i = 1:numOfLayers
        push!(myplot,layer(x=err[:,i],y=f_e[:,i],Geom.point,Geom.path,
        Theme(default_color=plotColors[i],default_point_size=pointSize)));
    end

    draw(PNG(fileName,plotSize[1],plotSize[2]),myplot)
    return nothing
end
```

3.2 ArenPrecisionTest

```
# Check if all the packages are installed or not
cond = "Gadfly" in keys(Pkg.installed()) &&
    "Colors" in keys(Pkg.installed()) &&
"Winston" in keys(Pkg.installed());
@assert cond "Please check if the following package(s) are installed:\
    Gadfly\
    Colors\
    ODEInterface"

# Load all the required packages
using Gadfly
using Colors
using ODEInterface
@ODEInterface.import_huge
loadODESolvers();

# Define the system of ODEs
function threebody(t,x,dx)
    mu = 0.012277471; ms = 1 - mu;
    r1 = vecnorm(x[1:2]-[-mu,0]);
    r2 = vecnorm(x[1:2]-[ ms,0]);

    dx[1] = x[3];
    dx[2] = x[4];
    dx[3] = x[1] + 2*x[4] - ms*(x[1]+mu)/r1^3 - mu*(x[1]-ms)/r2^3;
    dx[4] = x[2] - 2*x[3] - ms*     x[2]/r1^3 - mu*     x[2]/r2^3;

    return nothing
end

# Flag to check if all solvers were successful
printFlag = true;

# Initial conditions
t0 = 0.0; T = 17.06521656015796; x0=[0.994, 0.0, 0.0, -2.001585106379082];

# Get "reference solution"
opt = OptionsODE(OPT_EPS => 1.11e-16,OPT_RHS_CALLMODE => RHS_CALL_INSITU,
OPT_RTOL => 1e-16,OPT_ATOL=>1e-16);
(t,x_ref,retcode,stats) = dop853(threebody,t0, T, x0, opt);

if retcode != 1
    println("Reference solution failed")
else
    # Initialization for the loop
    # f_e = function evaluations
    f_e = zeros(Int32,89,3);
    # err = error for last step using infinity norm
    err = zeros(Float64,89,3);

    # solverNames = names of the solvers used for the plot
    solverNames = ["DOPRI5","DOP853","ODEX"];

    # Compute all the solutions
```

```
    for i=0:88

        # Set up the tolerance
        Tol = 10^(-3-i/8);

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

        # Solve using DOPRI5
        (t,x,retcode,stats) = dopri5(threebody,t0, T, x0, opt);
        # Check if solver was successful
        if retcode != 1
            printFlag = false;
            break;
        end
        f_e[i+1,1] = stats.vals[13];
        err[i+1,1] = norm(x_ref[1:2] - x[1:2],Inf);

        (t,x,retcode,stats) = dop853(threebody,t0, T, x0, opt);
        if retcode != 1
            printFlag = false;
            break;
        end
        f_e[i+1,2] = stats.vals[13];
        err[i+1,2] = norm(x_ref[1:2] - x[1:2],Inf);

        (t,x,retcode,stats) = odex(threebody,t0, T, x0, opt);
        if retcode != 1
            printFlag = false;
            break;
        end
        f_e[i+1,3] = stats.vals[13];
        err[i+1,3] = norm(x_ref[1:2] - x[1:2],Inf);
    end

    # Save the plot in PNG format
    if printFlag
        savePlotPNG("ArenstorfConvTest",f_e,err,solverNames);
    else
        println("Cannot generate plot due to solver failure.")
    end
end
```

3.3 RopePrecisionTest

```julia
# Check if all the packages are installed or not
cond = "Gadfly" in keys(Pkg.installed()) &&
"Colors" in keys(Pkg.installed()) &&
"ODEInterface" in keys(Pkg.installed());
@assert cond "Please check if the following package(s) are installed:\
    Gadfly\
    Colors\
    ODEInterface"

# Load all the required packages
using Gadfly
using Colors
using ODEInterface
@ODEInterface.import_huge
loadODESolvers();

# Number of subdivisions of the rope
global n = 40;

# Define the system of ODEs
function rope(t,x,dx)
    n2 = n*n; # n^2
    n3by4 = convert(Int64,3*n/4); # 3*n/4

    # Force in x-direction
    Fx = 0.4;
    # Force in y-direction
    Fy = cosh(4*t-2.5)^(-4);

    # Compute required matrices
    c = -cos(x[1:n-1]-x[2:n]);
    cDiag = [1;2*ones(n-2);3];
    C = spdiagm((c,cDiag,c),(-1,0,1));

    d = -sin(x[1:n-1]-x[2:n]);
    D = spdiagm((-d,d),(-1,1));

    # Compute the inhomogeneous term
    v = -(n2+n/2-n*[1:n;]).*sin(x[1:n])-n2*sin(x[1:n])*Fx;
    v[1:n3by4] = v[1:n3by4] + n2*cos(x[1:n3by4])*Fy;

    w = D*v+x[n+1:2*n].^2;
    u = C\w;

    # Write down the system
    dx[1:n] = x[n+1:2*n];
    dx[n+1:2*n] = C*v + D*u;

    return nothing
end

# Initial Conditions
t0 = 0.0; T = 3.723; x0=zeros(2*n);
```

```
# Compute the "reference solution"
opt = OptionsODE(OPT_EPS => 1.11e-16,OPT_RHS_CALLMODE => RHS_CALL_INSITU,
OPT_RTOL => 1e-16,OPT_ATOL=>1e-16);
(t,x_ref,retcode,stats) = dop853(rope,t0, T, x0, opt);

if retcode != 1
    println("Reference solution failed")
else
    # Initialization for the loop
    # f_e = function evaluations
    f_e = zeros(Int32,89,3);
    # err = error for last step using infinity norm
    err = zeros(Float64,89,3);

    # solverNames = names of the solvers used for the plot
    solverNames = ["DOPRI5","DOP853","ODEX"];

    # Compute all the solutions
    for i=0:88

        # Set up the tolerance
        Tol = 10^(-3-i/8);

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

        # Solve using DOPRI5
        (t,x,retcode,stats) = dopri5(rope,t0, T, x0, opt);
        # Check if solver was successful
        if retcode != 1
            printFlag = false;
            break;
        end
        f_e[i+1,1] = stats.vals[13];
        err[i+1,1] = norm(x_accurate[1:n] - x[1:n],Inf);

        # Solve using DOP853
        (t,x,retcode,stats) = dop853(rope,t0, T, x0, opt);
        # Check if solver was successful
        if retcode != 1
            printFlag = false;
            break;
        end
        f_e[i+1,2] = stats.vals[13];
        err[i+1,2] = norm(x_accurate[1:n] - x[1:n],Inf);

        # Solve using ODEX
        (t,x,retcode,stats) = odex(rope,t0, T, x0, opt);
        # Check if solver was successful
        if retcode != 1
            printFlag = false;
            break;
        end
```

```
        f_e[i+1,3] = stats.vals[13];
        err[i+1,3] = norm(x_accurate[1:n] - x[1:n],Inf);
    end

    # Save the plot in PNG format
    if printFlag
        savePlotPNG("RopeConvTest",f_e,err,solverNames);
    else
        println("Cannot generate plot due to solver failure")
    end
end
```

3.4 vdpolPrecisionTest

```julia
# Check if all the packages are installed or not
cond = "Gadfly" in keys(Pkg.installed()) &&
"Colors" in keys(Pkg.installed()) &&
"ODEInterface" in keys(Pkg.installed()) &&
"ForwardDiff" in keys(Pkg.installed());
@assert cond "Please check if the following package(s) are installed:\
    Gadfly\
    Colors\
    ODEInterface\
    ForwardDiff"

# Load all the required packages
using Gadfly
using Colors
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" from
# http://www.unige.ch/~hairer/testset/testset.html
f = open("vdpolRefSol.txt");
lines = readlines(f);

x_ref = Array{Float64}(11);

tmp = Array{Float64}(22);
counter = 1;
for l in lines
    tmp[counter] = parse(Float64,l);
```

```
    counter += 1;
end

x_ref = tmp[1:2:end];

close(f)

# 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);
# flops = Floating point operations
flops = zeros(33,3);

# Weights for computing flops
dim = 2; # dimension of the system
flopsRHS = 5; # Counted
flopsLU = ceil(2*((dim)^3)/3); # As per LU algorithm (can be less)
flopsFW_BW = (dim)^2; # As per FW/BW algorithm (can be less)
flopsJac = ceil(1.5*flopsRHS); # A guess at the moment

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);

    # Store the stats of the last t_end
    # for computing flops
    stats = Dict{ASCIIString,Any};

    # 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);
flops[i+1,1] = stats["no_rhs_calls"]*flopsRHS+
               stats["no_fw_bw_subst"]*flopsFW_BW +
               stats["no_jac_calls"]*flopsJac+
               stats["no_lu_decomp"]*flopsLU;

# 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);
flops[i+1,2] = stats["no_rhs_calls"]*flopsRHS+
               stats["no_fw_bw_subst"]*flopsFW_BW +
               stats["no_jac_calls"]*flopsJac+
               stats["no_lu_decomp"]*flopsLU;

# 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);
```

```
    flops[i+1,3] = stats["no_rhs_calls"]*flopsRHS+
                   stats["no_fw_bw_subst"]*flopsFW_BW +
                   stats["no_jac_calls"]*flopsJac+
                   stats["no_lu_decomp"]*flopsLU;
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
```

3.5 RoberPrecisionTest

```
# Check if all the packages are installed or not
cond = "Gadfly" in keys(Pkg.installed()) &&
"Colors" in keys(Pkg.installed()) &&
"ODEInterface" in keys(Pkg.installed()) &&
"ForwardDiff" in keys(Pkg.installed());
@assert cond "Please check if the following package(s) are installed:\
    Gadfly\
    Colors\
    ODEInterface\
    ForwardDiff"

# Load all the required packages
using Gadfly
using Colors
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] = 3e7*(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" from
# http://www.unige.ch/~hairer/testset/testset.html
f = open("roberRefSol.txt")
lines = readlines(f)
numLines = length(lines)
lenArray = convert(Int64,numLines/3)
```

```
x_ref = Array{Float64}(lenArray,3);
tmp = Array{Float64}(numLines)

counter = 1
for l in lines
    tmp[counter] = parse(Float64,l);
    counter +=1;
end

x_ref = [tmp[1:3:end] tmp[2:3:end] tmp[3:3:end]];

close(f)

# 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);
# flops = Floating point operations
flops = zeros(33,3);

# Weights for computing flops
dim = 3; # dimension of the system
flopsRHS = 13; # Counted
flopsLU = ceil(2*((dim)^3)/3); # As per LU algorithm (can be less)
flopsFW_BW = (dim)^2; # As per FW/BW algorithm (can be less)
flopsJac = ceil(1.5*flopsRHS); # A guess at the moment

# 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);

    # Store the stats of the last t_end
    # for computing flops
    stats = Dict{ASCIIString,Any};

    # 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);
flops[m+1,1] = stats["no_rhs_calls"]*flopsRHS+
               stats["no_fw_bw_subst"]*flopsFW_BW +
               stats["no_jac_calls"]*flopsJac+
               stats["no_lu_decomp"]*flopsLU;

# 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);
flops[m+1,2] = stats["no_rhs_calls"]*flopsRHS+
               stats["no_fw_bw_subst"]*flopsFW_BW +
               stats["no_jac_calls"]*flopsJac+
               stats["no_lu_decomp"]*flopsLU;

# 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);
    flops[m+1,3] = stats["no_rhs_calls"]*flopsRHS+
                   stats["no_fw_bw_subst"]*flopsFW_BW +
                   stats["no_jac_calls"]*flopsJac+
                   stats["no_lu_decomp"]*flopsLU;
end

if printFlag
    savePlotPNG("RoberPrecisionTest",f_e,err,solverNames);
else
    println("Plot cannot be generated due to failure");
end
```

References

[1] Ernst Hairer, Syvert P. Nørsett, Gerhard Wanner. *Solving Ordinary Differential Equations I - Nonstiff Problems*. Springer-Verlag, Heidelberg, 1993.

[2] Ernst Hairer, Gerhard Wanner. *Solving Ordinary Differential Equations II - Stiff and Differential-Algebraic Problems*. Springer-Verlag, Heidelberg, 1996.

[3] http://dcjones.github.io/Gadfly.jl/

[4] https://github.com/luchr/ODEInterface.jl/

[5] https://github.com/JuliaLang/ODE.jl

[6] https://github.com/sonVishal/ODEInterface.jl/

[7] http://www.unige.ch/ hairer/testset/testset.html