Implicit Euler's Method Demo

Kurt Bryan and SIMIODE

```
> restart;
  with(LinearAlgebra) : with(plots) :
```

Routine and demos for the implicit Euler method, for both a scalar ODE and a system.

DO THIS: Before executing the **Scalar Case Demo** or **System Case Demo**, go down and execute the block **Implicit Euler Routine** to define the routine "impeuler", then try the demo cases.

Scalar Case Demo: A demo of the scalar case x'(t) = -x(t) + t with initial condition x(t0) = x0, step size h = 0.1, solved out to time t = T with T = 1.

```
> t0 := 0 : x0 := 1 : T := 1.0 : h := 0.1 : #Vital parameters
```

A function to define the right side of the ODE x' = f(t,x)

```
> f(t, x) := -x + t
```

Call the routine "impeuler" (defined below); results in array "dat" at (t,y(t)) pairs.

```
> dat := impeuler(f, t0, T, x0, h) : N := nops([dat]) :
```

Plot the output

```
> plot([seq([dat[j][1], dat[j][2][1]], j = 1 ..N)], color = red, labels = [t, x])
```

System Case Demo:

First define initial time t0, initial vector x0, final time T, step size h.

```
> t0 := 0 : x0 := ⟨1, 0⟩ : T := 5.0 : h := 0.1;
```

Function to define ODE system x' = f(t,x).

```
> f(t, x) := ⟨x[2], -101·x[1] - 2·x[2]⟩
```

Solve the ODE numerically. Solution returned as pairs (t,x(t)) where x(t) is a vector.

```
> dat := impeuler(f, t0, T, x0, h) :
```

A parametric/phase plot of the solution

```
> N := nops([dat]) :
  plot([seq([dat[j][2][1], dat[j][2][2]], j = 1 ..N)], color = red, labels = [x[1], x[2]])
```

A plot of each component separately.

```
> p1 := plot([seq([dat[j][1], dat[j][2][1]], j = 1 ..N)], color = red) :
  p2 := plot([seq([dat[j][1], dat[j][2][2]], j = 1 ..N)], color = blue) :
  display(p1, p2, labels = [t, x])
```

Implicit Euler Routine: A routine to implement the implicit or backwards Euler method on x' = f(x,t) with x(t0) = x0. Here f(t,x) accepts a scalar "t", vector "x", initial time "t0", final time "T", initial data vector "x0", and step size "h". Here x0 can be a scalar in the 1D case. Output is a list [tk,xk] for t = t0 to t = T (or a bit farther than T if (T-t0)/h is not an integer) in steps of size h.

```
> impeuler := proc(f, t0, T, x0, h)
    local j, n, N, tk, xk, k, W, eq, sol, eqs, ff, m, soldis, i;

    #Bookkeeping for implicit Euler
```

```
N := ceil((T - t0)/h) :
tk := array(0..N) :
xk := array(0..N) :
tk[0] := t0 :

#Check x0, make it a column vector if necessary. Also, make f accept vector argument.
if not type(x0, 'Vector') then
  xk[0] := <x0> :
  ff := unapply(<f(t, x[1])>, t, x) :
else
  xk[0] := x0 :
  ff := f;
fi:
n := Dimension(xk[0]); #Number of equations or variables
#March solution out in time, use fsolve on equations
W := Vector(n, symbol='w') :
for k from 1 to N do
  tk[k] := tk[k - 1] + h :
  eq := (W - xk[k - 1])/h - ff(tk[k], W) :
  eqs := {seq(eq[j], j = 1..n)};
  sol := fsolve(eqs, {seq(W[j] = xk[k - 1][j], j = 1..n)}) :
   #Start with previous xk[k-1] as an initial guess
  m := nops([sol]) :
  if m > 1 then
     soldis := map(u -> abs(subs(u, W[1]) - xk[k - 1][1]), [sol]) :
     i := min[index](soldis) :
     sol := sol[i] :
  end:
  xk[k] := subs(sol, W) :
od:
seq([tk[j], xk[j]], j = 0..N);
end:
>
```