In [9]:

```
# Set up environment with correct dependencies
using Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
 Activating environment at `~/GitHub/MathSys/teaching/MA934/MA934-slid
es/Project.toml`
```

In [10]:

```
using Plots
using LaTeXStrings
using Revise
using LinearAlgebra
using DifferentialEquations
pyplot()
# Set default fonts for all plots
fnt = Plots.font("DejaVu Sans", 5.0)
default(titlefont=fnt, guidefont=fnt, tickfont=fnt, legendfont=fnt)
```

MA934

Solving equations

Root-finding in 1D

Task: find x such that $f(x) = 0$.

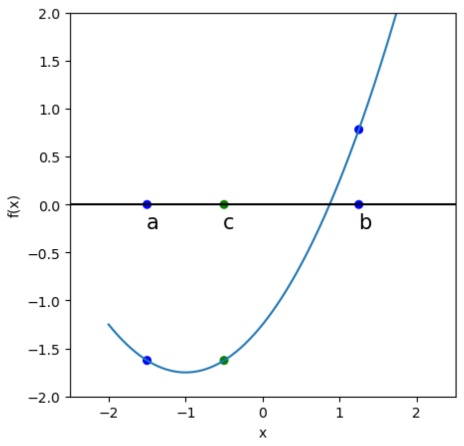
Interval $[a_0, b_0]$ brackets a root of $f(x)$ if $f(a_0)\, f(b_0) < 0$.

Bracket and bisect method:

1 $c_n = \frac{1}{2}(a_n + b_n)$
2 If $f(a_n)\, f(c_n) < 0$, $[a_{n+1}, b_{n+1}] = [a_n, c_n]$. Otherwise $[a_{n+1}, b_{n+1}] = [c_n, b_n]$
3 Repeat until $\epsilon_n = b_n - a_n < \epsilon_{\rm tol}$.

In [11]:

```
include("files/code/figures.jl")
figures.plot_bracket_root()
```



```
WARNING: replacing module fi
gures.
```

Out[11]:

```
(PyPlot.Figure(PyObject <Fig
```

Convergence: $\epsilon_n \sim 2^{-n}$.

Root-finding in 1D: Newton-Raphson iteration

Suppose root is at $x = x^*$ and current estimate is x_n.

Write $x_{n+1} = x_n + \delta_n$ and try to choose δ_n such that $f(x_{n+1}) = 0$:

Taylor expand:

$$0 = f(x_{n+1}) = f(x_n + \delta_n) = f(x_n) + \delta_n\, f'(x_n) + \mathcal{O}(\delta_n^2)$$

Assuming we are near the root, we neglect the $\mathcal{O}(\delta_n^2)$ terms and solve for δ_n:

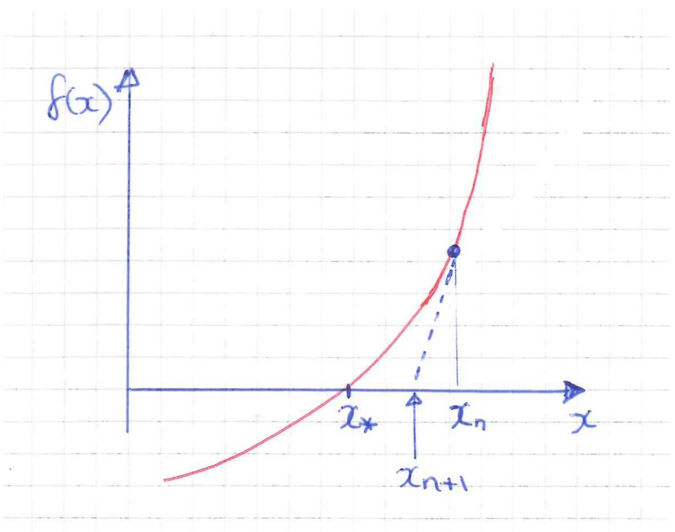
$$\delta_n = -\frac{f(x_n)}{f'(x_n)}.$$

This gives the Newton-Raphson iterative method:

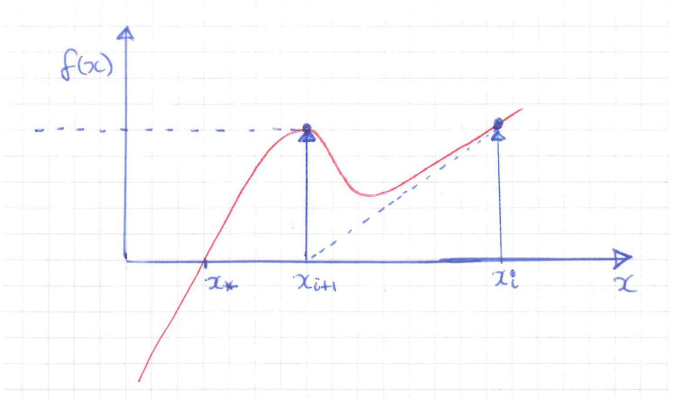
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

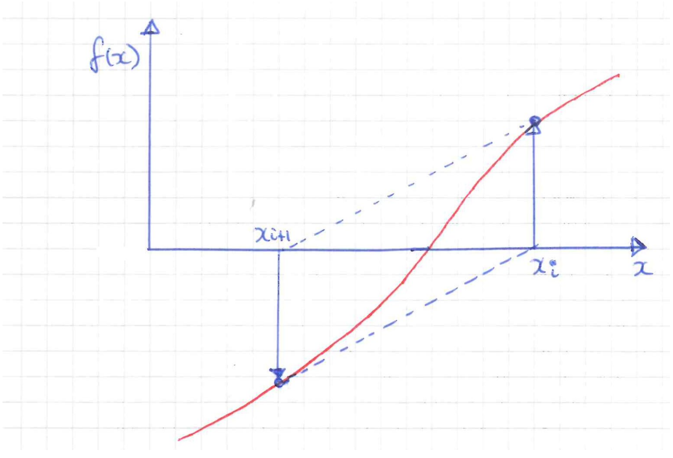
Convergence properties of Newton-Raphson

Geometrical interpretation:



NR is not guaranteed to converge:





Super-exponential convergence of Newton-Raphson

When NR does converge, it converges super-exponentially.

Let $\epsilon_n = x_n - x_*$. Then NR formula gives:

$$\epsilon_{n+1} = \epsilon_n - \frac{f(x^* + \epsilon_n)}{f'(x^* + \epsilon_n)}.$$

Taylor expanding and using $f(x^*) = 0$ one obtains (check):

$$f(x^* + \epsilon_n) = \epsilon_n\, f'(x_*)\left(1 + \frac{\epsilon_n}{2}\frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_n^3),$$

$$f'(x^* + \epsilon_n) = f'(x_*)\left(1 + \epsilon_n \frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_n^2).$$

Super-exponential convergence of Newton-Raphson

Keeping only the leading order terms on the RHS, there is a cancellation at $\mathcal{O}(\epsilon_n)$ and we get

$$\epsilon_{n+1} = \alpha\, \epsilon_n^2.$$

where $\alpha = \frac{f''(x_*)}{2\, f'(x_*)}$ depends on the properties of f at the root, x^*.

This nonlinear recursion relation can be linearised by $a_n = \log_\alpha \epsilon_n$ to give:

$$a_{n+1} = 2\, a_n + 1$$

Solution is $a_n = c_0\, 2^{n-1} - 1$ where c_0 is a constant.
In original variables, choosing c_0 to match the initial condition, ϵ_0, we get

$$\epsilon_n = \epsilon_0\, |\alpha|^{-1}\, |\alpha|^{2^n}.$$

Newton-Raphson in $\mathbb{R}^n$

System of n equations in $\mathbb{R}^n$:

$$\begin{aligned} F_1(x_1 \dots x_n) &= 0 \\ \vdots \quad & \quad \vdots \\ F_n(x_1 \dots x_n) &= 0. \end{aligned}$$

or

$$\mathbf{F}(\mathbf{x}) = 0.$$

As before, write $\mathbf{x}_{n+1} = \mathbf{x}_n + \boldsymbol{\delta}_n$ and try to choose $\boldsymbol{\delta}_n$ such that $\mathbf{F}(\mathbf{x}_{n+1}) = 0$.

Taylor expand:

$$\begin{aligned} 0 = \mathbf{F}(\mathbf{x}_{n+1}) &= \mathbf{F}(\mathbf{x}_n + \boldsymbol{\delta}_n) \\ &= \mathbf{F}(\mathbf{x}_n) + \mathbf{J}(\mathbf{x}_n)\, \boldsymbol{\delta}_n + \mathcal{O}(|\boldsymbol{\delta}|_n^2) \end{aligned},$$

where $\mathbf{J}(\mathbf{x}_n)$ is the Jacobian matrix

$$\mathbf{J}_{i\,j} = \frac{\partial F_i}{\partial x_j}(\mathbf{x}_n).$$

Neglecting the $\mathcal{O}(|\boldsymbol{\delta}_n|^2)$ terms, we obtain $\boldsymbol{\delta}_n$ from a linear solve:

$$\boldsymbol{\delta}_n = -\mathbf{J}(\mathbf{x}_n) \setminus \mathbf{F}(\mathbf{x}_n)$$

Example: Newton-Raphson iteration in $\mathbb{C}$.

For complex valued functions of a complex variable:

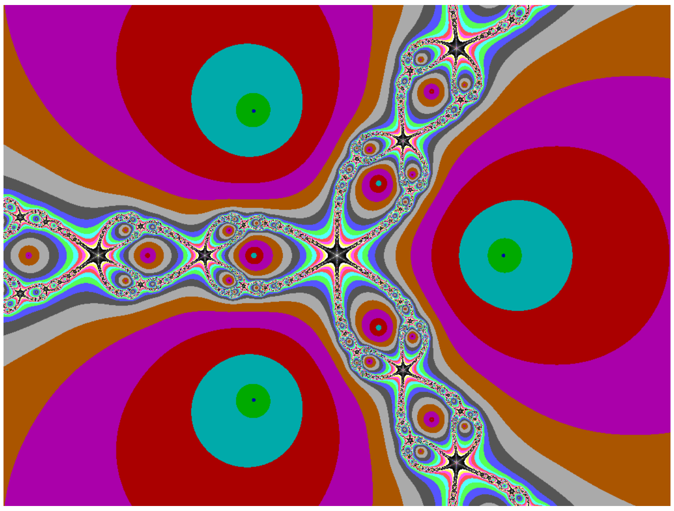
$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}$$

Nonlinear iterated map which can lead to very rich dynamics (periodic cycles, chaos, intermittency etc).

Example: compute basins of attraction in $\mathbb{C}$ of the roots of the polynomial

$$f(z) = z^3 - 1 = 0.$$

under NR iteration.



Ordinary differential equations

For initial value problems, it is generally sufficient to develop algorithms to solve autonomous first order systems in $\mathbb{R}^n$:

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u}) \quad \text{with} \quad \mathbf{u}(0) = \mathbf{U}_0.$$

For example, the second order non-autonomous equation, For example, the second order non-autonomous equation,

$$\frac{d^2 y}{dt^2} + 2\nu\omega \frac{dy}{dt} + \omega^2 y = F(t),$$

is equivalent (check) to this 3-dimensional autonomous system:

$$\frac{d}{dt}\begin{pmatrix} u^{(1)} \\ u^{(2)} \\ u^{(3)} \end{pmatrix} = \begin{pmatrix} -2\nu\omega u^{(1)} - \omega^2 u^{(2)} + F(u^{(3)}) \\ u^{(1)} \\ 1 \end{pmatrix} \quad \text{where} \quad \begin{pmatrix} u^{(1)} \\ u^{(2)} \\ u^{(3)} \end{pmatrix} = \begin{pmatrix} \frac{dy}{dt} \\ y \\ t \end{pmatrix}.$$

Discretisation and time stepping

Discretisation: approximate continuous $\mathbf{u}(t)$ on $t \in [0, T]$ by $\{\mathbf{u}_i \ : \ i = 0 \ldots N\}$. Here

$$\begin{aligned} \mathbf{u}_i &= \mathbf{u}(t_i) \\ t_i &= i\, h, \end{aligned}$$

and the "time step" is

$$h = \frac{T}{N}.$$

If h is "small" then

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u}) \ \text{ with } \ \mathbf{u}(0) = \mathbf{U}_0.$$

can be hueristically approximated by

$$\begin{aligned} \frac{\mathbf{u}_{i+1} - \mathbf{u}_i}{h} &= \mathbf{F}(\mathbf{u}_i) \text{ with } \mathbf{u}_0 \\ &= \mathbf{U}_0. \end{aligned}$$

Re-arranging, we get the (Forward Euler) time-stepping algorithm

$$\begin{aligned} \mathbf{u}_{i+1} &= \mathbf{u}_i + h\,\mathbf{F}_i \text{ with } \mathbf{u}_0 \\ &= \mathbf{U}_0, \end{aligned}$$

where $\mathbf{F}_i$ means $\mathbf{F}(\mathbf{u}_i)$.

Taylor's theorem

If $f(x)$ is a real-valued function which is differentiable $n+1$ times on the interval $[x, x+h]$ then there exists a point, ξ, in $[x, x+h]$ such that

$$\begin{aligned} f(x+h) = f(x) &+ \frac{1}{1!}\, h\, \frac{df}{dx}(x) + \frac{1}{2!}\, h^2\, \frac{d^2 f}{dx^2}(x) + \ldots \\ &+ \frac{1}{n!}\, h^n\, \frac{d^n f}{dx^n}(x) + h^{n+1} R_{n+1}(\xi) \end{aligned}$$

where

$$R_{n+1}(\xi) = \frac{1}{(n+1)!}\, \frac{d^{n+1} f}{dx^{n+1}}(\xi).$$

Useful for systematic analysis of discrete approximations to derivatives and ODEs.

Finite difference approximations of derivatives

Approximate $f'(x)$ by linear combinations of values of f at nearby points.
Taylor with $n = 1$:

$$f(x+h) = f(x) + h\, f'(x) + \mathcal{O}(h^2).$$

Using same discrete notation as before, rearrange to get:

$$f'(x_i) = \frac{f_{i+1} - f_i}{h} + \mathcal{O}(h). \qquad \text{Forward difference formula.}$$

Could also have started from $f(x-h)$ to obtain:

$$f'(x_i) = \frac{f_i - f_{i-1}}{h} + \mathcal{O}(h). \qquad \text{Backwards difference formula.}$$

In both cases, the approximation error is $\mathcal{O}(h)$.

Higher order finite differences: improving the error

Improved accuracy can be obtained by linearly combining more points:

$$\begin{aligned} f_{i+1} &= f_i + h\, f'(x_i) + \frac{1}{2}\, h^2\, f''(x_i) + \mathcal{O}(h^3) \\ f_{i-1} &= f_i - h\, f'(x_i) + \frac{1}{2}\, h^2\, f''(x_i) + \mathcal{O}(h^3). \end{aligned}$$

Take the linear combination $a_1\, f_{i-1} + a_2\, f_i + a_3\, f_{i+1}$:

$$\begin{aligned} a_1\, f_{i-1} + a_2\, f_i + a_3\, f_{i+1} = (a_1 + a_2 + a_3)\, f_i + (a_3 - a_1)\, h\, f'(x_i) \\ + \frac{1}{2}\,(a_3 + a_1)\, h^2\, f''(x_i) + O(h^3). \end{aligned}$$

We want to choose the a's to cancel the terms proportional to h^0 and h^2.

Higher order finite differences: improving the error

a_1 , a_2 and a_3 should therefore satisfy the equations

$$\begin{aligned} a_1 + a_2 + a_3 &= 0 \\ a_3 - a_1 &= 1 \\ a_3 + a_1 &= 0. \end{aligned}$$

We get $a_1 = -\frac{1}{2}$, $a_2 = 0$ and $a_3 = \frac{1}{2}$.

Rearranging to get an expression for $f'(x_i)$:

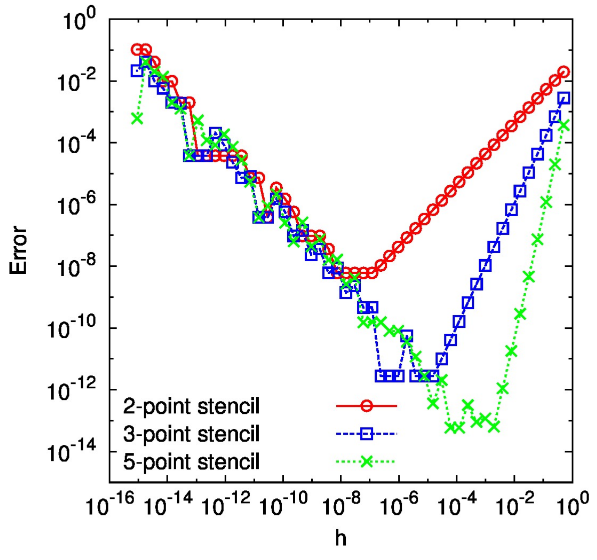
$$f'(x_i) = \frac{f_{i+1} - f_{i-1}}{2\,h} + O(h^2). \qquad \text{Centred difference formula.}$$

Higher order finite differences: improving the error

The set of points underpinning a finite-difference approximation is known as the "stencil".

A 5-point stencil leads to a 4th order accurate finite difference formula for $f'(x_i)$:

$$\frac{-f_{i+2}+8f_{i+1}-8f_{i-1}+f_{i-2}}{12\,h}+O(h^4).$$



Error as a function of h for several finite difference approximations to the derivative of $f(x) = \sqrt{x}$ at $x = 2$

Euler method again

Return to the ODE (assumed scalar from now on)

$$\frac{du}{dt} = F(u) \quad \text{with} \quad u(0) = U_0.$$

Accounting for the error in the forward difference approximation,

$$\frac{u_{i+1} - u_i}{h} + \mathcal{O}(h) = F(u_i) \;\text{ with }\; u_0 = U_0.$$

The Forward Euler time-stepping algorithm thus has *step-wise error* $\mathcal{O}(h^2)$:

$$u_{i+1} = u_i + h\, F_i + \mathcal{O}(h^2) \;\text{ with }\; \mathbf{u}_0 = \mathbf{U}_0,$$

Global error is $\mathcal{O}(N h^2) = T\, \mathcal{O}(h)$ since $h = \frac{T}{N}$.

Implicit methods: Backward Euler

We could equally have used the backward difference approximation:

$$\frac{u_i - u_{i-1}}{h} + \mathcal{O}(h) = F(u_i) \;\text{ with }\; u_0 = U_0.$$

which gives (with $i \to i+1$)

$$u_{i+1} = u_i + h\, F_{i+1} + O(h^2).$$

This is called the Backward Euler method. Note u_{i+1} appears on both sides.

This is an example of an *implicit* method. Implicit methods require a Newton-Raphson solve at each timestep.

Improving accuracy: trapezoidal method

Another way to think about time-stepping algorithms is via approximation of

$$u(t+h) = u(t) + \int_t^{t+h} F(u(\tau))\, d\tau.$$

Approximation of integral with left and right Riemann rule gives the forward and backward Euler methods respectively (check).

If we use the Trapezoidal rule we get:

$$u_{i+1} = u_i + \frac{h}{2}\left[F_i + F_{i+1}\right].$$

This is called the implicit trapezoidal method.

The implicit trapezoidal method improves on the Euler methods because it has a stepwise error of $\mathcal{O}(h^3)$. How to show this?

Error analysis of timestepping rules

Consider Taylor expansion of u_{i+1}:

$$\begin{aligned} u_{i+1} &= u_i + h\frac{du}{dt}(t_i) + \frac{h^2}{2}\frac{d^2u}{dt^2}(t_i) + O(h^3) \\ &= u_i + hF_i + \frac{1}{2}h^2 F_i\, F_i' + O(h^3) \end{aligned}$$

Note the use of the chain rule:

$$\begin{aligned} \frac{d^2u}{dt^2}(t_i) &= \frac{d}{dt}F(u(t))\Big|_{t=t_i} = F'(u(t))\,\frac{du}{dt}\Big|_{t=t_i} \\ &= F'(u(t))\,F(u(t))\Big|_{t=t_i} = F_i'\,F_i. \end{aligned}$$

Error analysis of timestepping rules

Idea: Taylor expand RHS of timestepping rule and identify the first order at which expansion differs from the above.

$$\begin{aligned} F(u(t_{i+1})) &= F(u_i + hF_i + \mathcal{O}(h^2)) \\ &= F(u_i) + (hF_i)\frac{dF}{du}(u(t_i)) + O(h^2) \\ &= F_i + hF_iF_i' + O(h^2). \end{aligned}$$

RHS of implicit trapezoidal method is therefore

$$u_i + \frac{h}{2}[F_i + F_{i+1}] = u_i + hF_i + \frac{1}{2}h^2 F_i\, F_i' + \mathcal{O}(h^3)$$

Comparing with u_{i+1}, stepwise error is $\mathcal{O}(h^3)$.

Predictor-corrector methods

Can we get higher order accuracy with an explicit scheme?

Predictor-corrector idea: use a less accurate explicit method to predict u_{i+1} and then use the higher order formula to correct this prediction.

Example: improved Euler method:

1. Use forward Euler as predictor:
$$u_{i+1}^* = u_i + hF_i,$$
and calculate
$$F_{i+1}^* = F(u_{i+1}^*).$$
2. Use the Trapezoidal Method to correct this :
$$u_{i+1} = u_i + \frac{h}{2}\left[F_i + F_{i+1}^*\right].$$

Error analysis (similar to above - check) shows stepwise error is $\mathcal{O}(h^3)$.

Choosing the timestep

In practice we need to operate at a finite value of h. How do we choose it?

Measure the error by comparing the numerical solution at a grid point, $\widetilde{u}_i$, to the exact solution, $u(t_i)$.

Two tolerance criteria are commonly used:

$$\begin{aligned} E_a(h) &= |\widetilde{u}_i - u_i| \le \epsilon \qquad \text{Absolute error threshold,} \\ E_r(h) &= \frac{|\widetilde{u}_i - u_i|}{|u_i|} \le \epsilon \qquad \text{Relative error threshold.} \end{aligned}$$

Fixing error tolerance allows us to take larger timesteps when the solution is varying slowly.

Adaptive timestepping : estimating error

Problem: exact solution usually not known. Use trial steps:

1. Take a trial step with stepsize h from u_i to get estimate u_{i+1}^{B}.
2. Take 2 trial steps with stepsize $\frac{h}{2}$ from u_i to get estimate u_{i+1}^{S}.
3. Estimate of local error is
$$\Delta = \left|u_{i+1}^{\mathrm{B}} - u_{i+1}^{\mathrm{S}}\right|.$$

Adaptive timestepping : selecting the new stepsize

If timestepping method is n^{th} order, we know there is a constant c such that

$$c\, h_{\rm old}^n = \Delta.$$

For maximum efficiency, we should choose $h_{\rm new}$ to saturate the error threshold:

$$c\, h_{\rm new}^n = \epsilon.$$

Eliminating c between these two equations gives:

$$h_{\rm new} = \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h_{\rm old}$$

Stiff problems

With adaptive stepsizing we expect to take larger timesteps when the solution is varying slowly and smaller timesteps when the solution is varying rapidly.

Sometimes explicit methods result in very small step sizes even when the solution is varying *slowly*. Such problems are said to be *stiff*.

Efficient solution of stiff problems requires bespoke stiff solvers that are usually implicit.

Stiff problems

Computational stiffness is a complicated topic: depends on the equation, the initial condition, the numerical method being used and the interval of integration. See this article (http://www.scholarpedia.org/article/Stiff_systems).

Common feature is that the solution is varying slowly but "nearby" solutions vary rapidly.

The simple equation

$$\frac{du}{dt} = -\lambda\, u \quad \text{with } u(0) = 1,$$

turns out to have this property when $\lambda \gg 1$ or solution interval is long.

Solving ODEs in Julia

DifferentialEquations.jl (https://docs.juliadiffeq.org/stable/) is a well developed system for solving systems of differential equations. In its basic form, it is very simple to use.

Let's solve the *relaxation oscillator* equations:

$$\frac{dx}{dt} = \mu(y - (\frac{1}{3}x^3 - x))$$
$$\frac{dy}{dt} = -\frac{1}{\mu}x.$$

Steps:

1. Define the RHS of the system of equations.
2. Define an `ODEProblem` object.
3. Integrate the `ODEProblem` .
4. Plot and analyse the solution.

Step 1: define the RHS

- `du` is the right hand side of the system (as a vector)
- `u` are the dependent variables for (as a vector)

- `t` is the time variable (for non-autonomous systems)
- `p` is a list of parameters that need to be passed in

In [12]:

```
using DifferentialEquations

function rel_osc!(du,u,p,t)
    x,y = u
    μ = p[1]
    du[1] = μ *(y - ((x^3.0)/3.0 - x
    du[2] = - x / μ
end
```

Out[12]:

```
rel_osc! (generic function w
ith 1 method)
```

Step 2: define the problem

- specify initial condition, `u0` .
- specify solution interval, `tspan`
- provide values for the parameters, `p`
- create an `ODEProblem` object. Takes `u0` , `tspan` , `p` and the RHS function from step 1 as arguments.

In [13]:

```
u0 = [-0.1,-0.1]
tspan = (0.0,25.0)
p = [1.0]
prob = ODEProblem(rel_osc!,u0,tspan,p
```

Out[13]:

```
ODEProblem with uType Array
{Float64,1} and tType Float6
4. In-place: true
timespan: (0.0, 25.0)
u0: [-0.1, -0.1]
```

Step 3: integrate the equations

In [14]:

```
sol=solve(prob)
```

Out[14]:

```
retcode: Success
Interpolation: automatic order switching interpolation
t: 53-element Array{Float64,1}:
  0.0
  0.09145051250313663
  0.31375450889603507
  0.6010932493697769
  0.9258077125963625
  1.3358990905985302
  1.859136183968317
  2.4173888480363694
  3.0158683466370477
  3.664269044897806
  4.313308032049602
  4.981826684600039
  5.705263784525903
  ⋮
 20.16434710416285
 20.586204287015974
 20.97913832847751
 21.396411144140433
 21.959213865321267
 22.509942475022964
 23.067289772094828
 23.48132189093073
 23.91974436753072
 24.328822836404225
 24.75078835122298
 25.0
u: 53-element Array{Array{Float64,1},1}:
 [-0.1, -0.1]
 [-0.1186531999110431, -0.09000790901327195]
 [-0.1667839883593314, -0.058344936365905556]
 [-0.23228332829387688, -0.0010502728595811514]
 [-0.3042329692076237, 0.0862364365929560 9]
 [-0.3760612867629594, 0.22680687043439485]
 [-0.40019675748993805, 0.43415848944671503]
 [-0.29077220028517115, 0.6347390347933562]
 [0.03888491034505014, 0.7223756297244842]
 [0.6593114117452428, 0.5075376499564139]
 [1.219474331089501 8, -0.12326233724740758]
 [1.2405126550588144, -0.9780051049170592]
 [0.7081151474076964, -1.714916063771932]
 ⋮
 [-1.707988257535603, -1.5401773278308366]
 [-2.0082143446931555, -0.7329533612674098]
 [-1.90943150688298 1, 0.043006917588200445]
 [-1.6703228019127654, 0.7930173873295067]
 [-1.2088366710803153, 1.6113688887907913]
 [-0.5102251459883388, 2.100816295182093]
 [0.7060457256994783, 2.074574928907806]
 [1.6843418494014362, 1.564895097552747]
 [2.0082747002842334, 0.7289314472969659]
 [1.9010916033370566, -0.07761069031286436]
```

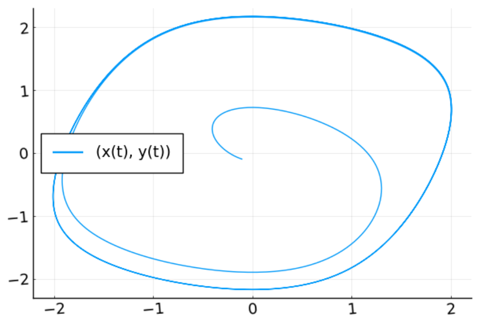
```
[1.6547613321841241, -0.8310187128826674]
```

Step 4: plot the solution

In [15]:

```
plot(sol,vars=(1,2), label="(x(t), y(
```

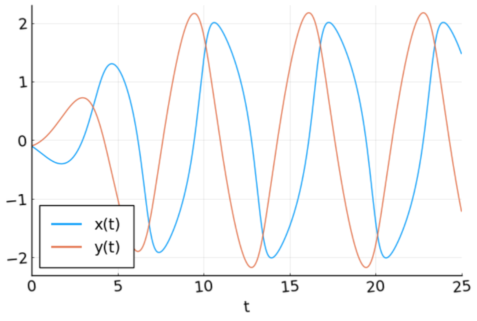
Out[15]:



In [16]:

```
plot(sol, vars=(0,1), label="x(t)")
plot!(sol,vars=(0,2), label="y(t)")
```

Out[16]:



Comparison of Forward and Backward Euler for decay equation

In [17]:

```
function exp_decay(u,p,t)
    λ = p[1]
    return -λ*u
end

u0 = 100.0; λ = 10.0;T=20.0/λ
h = 0.15
tspan = (0.0,T)
p = [λ]
prob = ODEProblem(exp_decay,u0,tspan,
```

Out[17]:

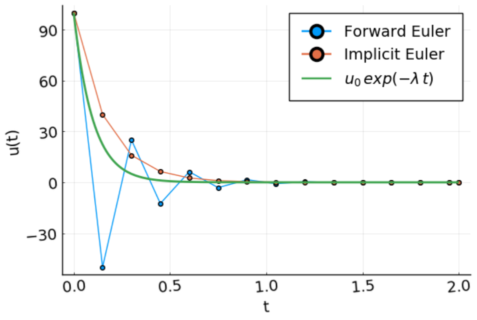
```
ODEProblem with uType Float6
4 and tType Float64. In-plac
e: false
timespan: (0.0, 2.0)
u0: 100.0
```

In [18]:

```
include("files/code/figures.jl")
sol1  = solve(prob, Euler(), dt=h)
sol2  = solve(prob, ImplicitEuler(),
figures.plot_Euler(sol1, sol2, λ, u0)
```

```
WARNING: replacing module fi
gures.
```

Out[18]:



Two timescales in relaxation oscillator

In [ ]:

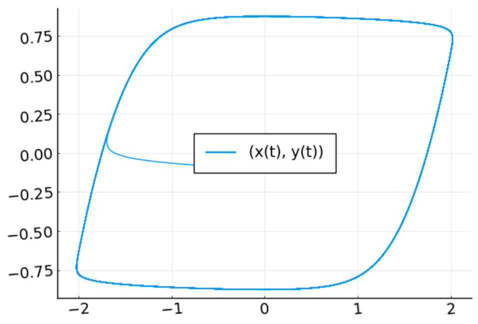
```

```

In [19]:

```
# Try changing μ
μ = 5.0
p1, p2 = figures.plot_relaxation_osci
p1
```

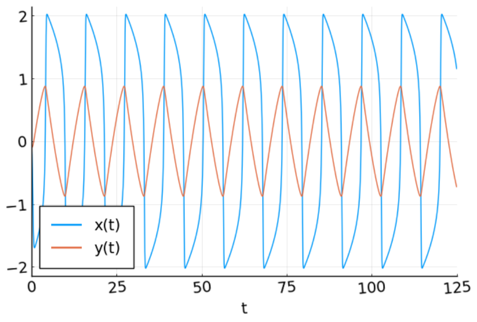
Out[19]:



In [20]:

```
p2
```

Out[20]:


x(t)
y(t)
2
1
0
−1
−2
0
25
50
75
100
125
t