```
// TestExp5.cpp
//
// dx/dt = - grad(U(x)) , grad == gradient
// Transform f(x) = 0 to a least-squares problem, U = f*f
//
// Tip iceberg: minimise a function using an ODE solver for a gradient system.
//
// compute z = exp(5); f(z) = z - exp(5); argmin f(z)*f(z)
//
// It's a sledgehammer but the method can be applied to constrained optimisation for ODE
systems.
// To be discussed elsewhere. It can be applied in all sorts of places.
// Current ML wisdom uses Gradient Descent (GD) method, which is Euler's method, and much
weaker in many
// ways to the approach taken here. It's much more robust and general than GD.
//
// Exercise:
// 1. generalise to ODE systems.
// 2. generalise to constrained optimisation and penalty method.
// 3. methods to compute gradient in n dimensions.
// 4. Use in an ANN instead of flaky GD.
//
// "The world is continuous, but the mind is discrete." David Mumford
// Natura non facit saltum (Nature does nothing in jumps)
//
// My take is that Gradient Descent is a fabrication; ODE gradient system are closer
// to the physical world.
//
// DJD
//
// Using ODEs for optimisation.
//
// http://www.unige.ch/~hairer/preprints/gradientflow.pdf
// https://authors.library.caltech.edu/26703/2/postscript.pdf
// https://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation/
//
https://www.dam.brown.edu/people/geman/Homepage/Image%20processing,%20image%20analysis,%2
0Markov%20random%20fields,%20and%20MCMC/Diffusions%20and%20optimization.pdf
// https://www.cs.ubc.ca/~ascher/papers/adhs.pdf
//
// Nesterov's method
// https://epubs.siam.org/doi/epdf/10.1137/21M1390037
//
//
// For training courses, see
//
// https://www.datasim.nl/onlinecourses/97/distance-learning-ordinary-and-partial-
differential-equations
//
//
https://www.datasim.nl/application/files/9015/4809/1157/DL_Ordinary_and_Partial_Different
ial_Equations.pdf
//
// https://www.datasim.nl/onlinecourses
//
// See also my book on ODE/PDE/FDM in computational finance
//
```

```
// https://www.wiley.com/en-
us/Numerical+Methods+in+Computational+Finance:+A+Partial+Differential+Equation+(PDE+FDM)+
Approach-p-9781119719670
//
// (C) Datasim Education BV 2018-2023
//
//


#include <iostream>
#include <vector>
#include <cmath>
#include <complex>


// https://www.boost.org/doc/libs/1_82_0/libs/numeric/odeint/doc/html/index.html
#include <boost/numeric/odeint.hpp>

// Preliminary notation
using value_type = double;
using state_type = std::vector<value_type>;

// Using C++11 functions
template <typename T>
	using FunctionType = std::function<T(const T& arg)>;
using CFunctionType = FunctionType<std::complex<value_type>>;


template <typename T>
	FunctionType<T> operator * (FunctionType<T>& f, FunctionType<T>& g)
{ // Multiplication (higher-order function)

	return [=](T x)
	{
		return  f(x)*g(x);
	};

}

// Complex Step Method for gradients (others: divided difference, AAD, analytic,..)
value_type CSM(const CFunctionType& f, value_type x, value_type h)

{ // df/dx at x using the Complex step method (Trapp/Squire)

	std::complex<value_type> z(x, h); // x + ih, i = sqrt(-1)
	return std::imag(f(z)) / h;
}

std::complex<double> ObjFunc(const std::complex<value_type>& z)
{ // My hard-coded specific function

	// compute z = exp(5); f(z) = z - exp(5); argmin f(z)*f(z)
	const std::complex<double> a(std::exp(5.0), 0.0); // => 148.413

	// Original function f(x) = 0 (Newton)
	//CFunctionType f = [&](const std::complex<double>& z) { return z - a; };
	CFunctionType f = [&](const std::complex<double>& z) { return z * z / 2.0 +
std::abs(z); };
```

```
	CFunctionType f2 = f * f;

	// Least squares function
	return f2(z);
}

// Free function to model RHS in dy/dt = RHS(t,y)
void Ode(const state_type &x, state_type &dxdt, const value_type t)
{
	//dxdt[0] = 2.0 * (x[0] - std::exp(5.0)); // Exact derivative
	const double h = 1.0e-156; // Small h but no overflow!
	dxdt[0] = CSM(ObjFunc, x[0], h);

	// Transform semi-infinite time interval (0, infinity) to (0,1)
	// Then we look at asymptotic behaviour..
	double tau = (1.0 - t) * (1.0 - t);
	dxdt[0] = -dxdt[0]/tau;
}

void print(std::string name, std::size_t steps, value_type v)
{
	std::cout << "Number of steps " << name << std::setprecision(16) << steps <<
"approximate: " << v << '\n';
}

int main()
{
    namespace Bode = boost::numeric::odeint;


    // Initial condition
	value_type L = 0.0;
	value_type T = 0.9998; // HEURISTIC simulates T = infinity
	value_type initialCondition = 1; // Convergence independent of IC? Wow
	value_type dt = 1.0e-5;

	{
		// Cash Karp, middle-of-road
		state_type x{ initialCondition };
		std::size_t steps = Bode::integrate(Ode, x, L, T, dt);
		print("Cash Karp ", steps, x[0]);
	}
	{
		// Bulirsch-Stoer method, high-class solver
		state_type x{ initialCondition};
		Bode::bulirsch_stoer<state_type, value_type> bsStepper;	//
O(variable), controlled stepper
		std::size_t steps = Bode::integrate_const(bsStepper, Ode, x, L, T, dt);
		print("Bulirsch-Stoer ", steps, x[0]);

	}

	return 0;
}
```