UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

ECE 204 *Numerical methods*

# Authoring an initial-value problem solver

Douglas Wilhelm Harder, LEL, M.Math.
dwharder@uwaterloo.ca
dwharder@gmail.com

# Introduction

- In this topic, we will
  - Look at the weaknesses of our implementations
  - Consider how to reduce the work for the user
  - Discuss how to use classes to provide better user interfaces

# Review of our IVP solvers

- The solvers we have implemented so far are classic procedural functions
  - They all return multiple arrays
  - The Matlab IVP solver routine `ode45(...)` is similar
  - The Maple `dsolve(...)` routine,
    however, returns a callable function

- All the user wants, however, is to evaluate the solution at various points
  - Passing back arrays requires the user to perform the interpolation or spline calculations

# A better approach

- Solution:
  - Create a class, where each instance is a solver for a particular initial-value problem
    - The description of the IVP is passed to the constructor
  - The class has one public member operator:

```
double operator()( double t )
vec<N> operator()( double t )
```

# A better approach

- All other data is kept internal inside the instance of the class
  - This includes the *t*-values and the approximations of the values and derivatives
- Question: How do we know how far to approximate to?
  - We don't, and we don't care, either
  - The user will create an instance:

```
int main() {
    ivp y{ f, 0.0, 1.0,
           std::make_pair( 0.0001, 0.01 ), 1e-4 };

    for ( unsigned int k{0}; k <= 1000; ++k ) {
        std::cout << y(0.01*k) << std::endl;
    }
    return 0;
}
```

# A better approach

- Do we make the calculation each time?
  - No
- Strategy:
  - Store the *t*-values and approximations in a `std::vector` member variable
  - If the user asks for an approximation at a specific *t*, check if lies between two approximations
    - If no, keep approximating from the last approximated *t*-value until you pass the requested time *t*
  - Any newly calculated approximations are appended to the `std::vector` member variable

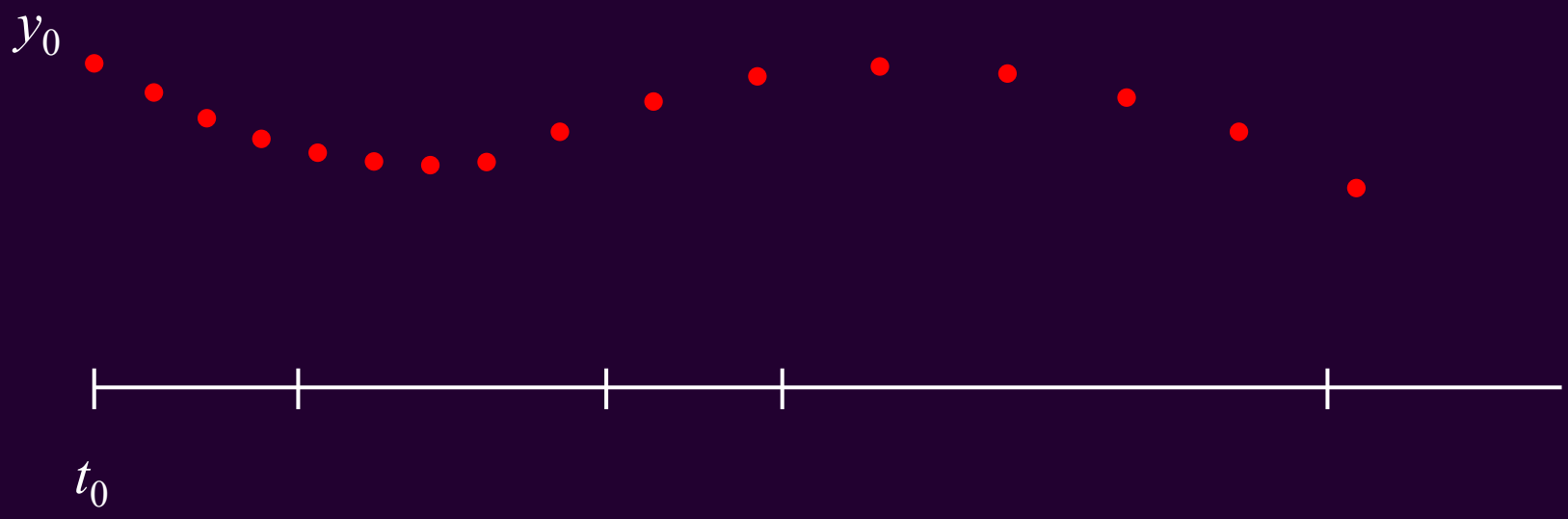
# Issue with naming…

- The standard template library class equivalent to an *array* on steroids is unfortunately called `std::vector`
  - This was an acknowledged mistake on the part of the designer of the STL, Alex Stepanov
- Issues:
  - You cannot perform vector addition, nor can you perform scalar multiplication
  - `std::vector` are variable in size: you can resize a `std::vector`

- Just remember:
  The `std::vector` class has absolutely nothing to do with vectors from linear algebra

# A better approach

- Internally, we proceed as follows:



# A better approach

- Thus, the user need know nothing about how these computations are being made
  - The user would look at the description and see if the package is appropriate in the state provided

# A better approach

- Consequently, the user can simply do:

```
int main() {
    ivp y{ f, 0.0, 1.0,
           std::make_pair( 0.0001, 0.01 ), 1e-4 };

    for ( unsigned int k{0}; k <= 1000; ++k ) {
        std::cout << y(0.01*k) << std::endl;
    }

    return 0;
}
```

  - Essentially all these values will be spline results
    - The user doesn't care

# Pseudo-code

```
double ivp::operator()( double t ) {
    if ( … ) {
        // If we have already approximated the solution up to or
        // beyond t, find the approximations on either side of t
        // in the std::vector and return the cubic spline at t
    } else {
        // Create a queue to store new approximations
        // Use the Dormand-Prince method to continue
        // approximating new t values and appending these
        // new approximations onto the queue, until
        // we have approximated a point at or beyond t

        // Expand the std::vector, and move the data from the
        // queue to the std::vector

        // Use a cubic spline at t
    }
}
```

# Summary

- Following this topic, you now
  - Understand how to make interfaces more user friendly
  - Are aware of the `ivp` class and how it would be implemented

# References

[1] https://en.wikipedia.org/wiki/Dormand-Prince_method

[3] https://en.wikipedia.org/wiki/Adaptive_algorithm

[4] https://en.wikipedia.org/wiki/Adaptive_step_size

# Acknowledgments

None so far.

# Colophon

These slides were prepared using the Cambria typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`. Mathematical equations are prepared in MathType by Design Science, Inc.

Examples may be formulated and checked using Maple by Maplesoft, Inc.

The photographs of flowers and a monarch butter appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens in October of 2017 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

# Disclaimer

These slides are provided for the ECE 204 *Numerical methods* course taught at the University of Waterloo. The material in it reflects the author's best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.