ECE 204 Project 4

Douglas Harder

March 9, 2024

1 Introduction

In this topic, we will approximate the solution to a system of three coupled initial-value problems (IVPs) using the 4th-order Runge Kutta method and cubic splines. The source code you can start with can be found at `https://replit.com/@dwharder/ECE-204-Project-4` .

2 Tuples

You will use the `vec<3>` class used in Project 1. Additionally, English has words such as as single, double, triple, quadruple, quintuple, sextuple, septuple, octuple, and nonuple to describe something consisting of 1 through 9 parts, respectively. The prevalence (though not universality) of the last five letters begin "tuple", the word used to describe something consisting of n parts is thus referred to, generally, as an n-tuple, and hence, while the data structure for returning a double is `std::pair` (for some odd reason, `std::double` would not work in C++). To access the member variables of a pair, you use `var_name.first` and `var_name.second`, while for a `std::tuple`, you use `std::get<k>( var_name )` where `k` must be specified at compilation time. For example:

```
#include <tuple>
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
  // The compiler determines that this is
  // a tuple of an int, a double and a bool
  // Note: 'auto' tells the compiler:
  //   "You figure out the type..."
  auto variable{ std::make_tuple( 3, 5.4, true ) };
  std::cout << std::get<0>( variable ) << ", "
            << std::get<1>( variable ) << ", "
            << std::get<2>( variable ) << std::endl;

  return 0;
}
```

The output is `3, 5.4, 1`.

3 The mathematics

Going onto the mathematics, suppose your system of differential equations was given by

$$\begin{aligned} u^{(1)}(t) &= u(t) + v(t) - 3w(t) + 1 \\ v^{(1)}(t) &= 2u(t) + v(t) - w(t) + 2 \\ w^{(1)}(t) &= u(t) + 4v(t) + w(t) - 1 \end{aligned} \tag{1}$$

This defines a system of three initial-value problems in three unknown functions. We could rewrite this as follows: Define

$$\mathbf{y}(t) = \begin{pmatrix} u(t) \\ v(t) \\ w(t) \end{pmatrix},$$

so that

$$\mathbf{y}^{(1)}(t) = \begin{pmatrix} u^{(1)}(t) \\ v^{(1)}(t) \\ w^{(1)}(t) \end{pmatrix},$$

and thus, we may define

$$\mathbf{y}^{(1)}(t) = \mathbf{f}(t, \mathbf{y}(t))$$

where

$$\mathbf{f}(t, \mathbf{y}(t)) = \begin{pmatrix} y_0(t) + y_1(t) - 3y_2(t) + 1 \\ 2y_0(t) + y_1(t) - y_2(t) + 2 \\ y_0(t) + 4y_1(t) + y_2(t) - 1 \end{pmatrix},$$

This would define a function in C++ as follows:

```
vec<3> f1( double t, vec<3> y ) {
    return vec<3>{
          y(0) +   y(1) - 3*y(2) + 1,
        2*y(0) +   y(1) -   y(2) + 2,
          y(0) + 4*y(1) -   y(2) - 1
    };
}
```

For example, if you were were to execute the statement

```
std::cout << f1( 0.3, { 1.0, 2.0, 3.0 } ) << std::endl;
```

then the output would be

```
[-5.000000 3.000000 5.000000]'
```

You will author two functions:

```
std::tuple<double *, vec<3> *, vec<3> *> rk4(
  std::function<vec<3>( double t, vec<3> y )> f,
  std::pair<double, double> t_rng, vec<3> y0,
  unsigned int n
);

vec<3> ivp_evaluate(
    double t,
    std::tuple<double *, vec<3> *, vec<3> *> y,
    unsigned int n
);
```

The first function will take the inputs that define a first-order initial-value problem together with the number of intervals, and using the Runge Kutta method, will return a tuple of three dynamically allocated arrays:

1. An array of $n+1$ t_k-values.
2. An array of $n+1$ vectors of $\mathbf{y}_k$-values where the first is the initial condition vector and the balance are approximations of $\mathbf{y}(t_k)$.
3. An array of $n+1$ vectors of $\mathbf{f}(t_k, \mathbf{y}_k)$, being the slopes calculated at $(t_k, \mathbf{y}_k)$.

The second function will use cubic splines to interpolate an approximation of $\mathbf{y}(t)$ for some value of $t_0 \leq t \leq t_f$ using data returned int the previous tuple.

4 The Runge Kutta algorithm

In this document, we will refer to the `first` and `second` member variables of the parameter `std::pair<double, double> t_rng` as t_0 and t_f, respectively.

You will proceed with an initial set-up as follows:

1. You will first determine the value $\Delta t = \frac{t_f - t_0}{n}$. You will then allocate an array `ts` of `double` of capacity $n + 1$ and assign to each entry of the array $t_0 + k\Delta t$.
2. Next, you will allocate an array `ys` of `vec<3>` of capacity $n + 1$ and assign to the first entry the initial vector $\mathbf{y}_0$.
3. Finally, you will allocate a second array `dys` also of `vec<3>` of capacity $n + 1$ and assign to the first entry the vector $\mathbf{f}(t_0, \mathbf{y}_0)$.

Next, you will proceed as follows: iterating from $k = 0$ to $n - 1$:

1. Given the values t_k, $\mathbf{y}_k$ and the various slopes ($\mathbf{f}(t_k, \mathbf{y}_k)$, $\mathbf{f}(t_k + \frac{1}{2}h, \mathbf{y}_k + \frac{1}{2}hs_0)$, etc.), you will use the Runge Kutta method to approximate the solution at $\mathbf{y}(t_{k+1})$ and assign that to `ys[k + 1]`.
2. Given this approximation of the solution, you will then calculate $\mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$ and assign that `dys[k + 1]`.

You will then return a tuple using `std::make_tuple( ts, ys, dys )`.

In case you did not determine this, the variable names each refers to an array of t-values, of $\mathbf{y}$-values, and the derivatives of the $\mathbf{y}$-values (or slopes).

Here is the source code for the version of the Runge Kutta algorithm described in class:

```
std::tuple<double *, double *, double *> rk4(
  std::function<double( double t, double y )> f,
  std::pair<double, double> t_rng, double y0,
  unsigned int n
) {
  assert( n > 0 );
  double h{ (t_rng.second - t_rng.first)/n };
  double  *ts{ new double[n + 1] };
  double  *ys{ new double[n + 1] };
  double *dys{ new double[n + 1] };

  // Assign the initial values
  ts[0] = t_rng.first;
  ys[0] = y0;
  dys[0] = f( ts[0], ys[0] );

  for ( unsigned int k{0}; k < n; ++k ) {
    ts[k + 1] = ts[0] + h*(k + 1);
    // Calculate the four slopes
    double s0{ dys[k] };
    double s1{ f( ts[k] + h/2.0, ys[k] + h*s0/2.0 ) };
    double s2{ f( ts[k] + h/2.0, ys[k] + h*s1/2.0 ) };
    double s3{ f( ts[k + 1],     ys[k] + h*s2 ) };

    // Estimate the next value using a convex
    // combination of the four slopes
    ys[k + 1] = ys[k] + h*(
      s0 + 2.0*s1 + 2.0*s2 + s3
    )/6.0;

    // Calculate the slope at (t   , y   )
    //                          k+1   k+1
    dys[k + 1] = f( ts[k + 1], ys[k + 1] );
  }

  return std::make_tuple( ts, ys, dys );
}
```

This takes a single initial-value problem, so

$$y^{(1)}(t) = f(t, y(t)) \text{ with } y(t_0) = y_0.$$

You need to update this code so that it solves a system of coupled initial-value problems

$$\mathbf{y}^{(1)}(t) = \mathbf{f}(t, \mathbf{y}(t)) \text{ with } \mathbf{y}(t_0) = \mathbf{y}_0.$$

5 The interpolation algorithm

You will write a second function that takes a t-value and the the output of the first to return an approximation of $\mathbf{y}(t)$ using splines:

```
vec<3> ivp_evaluate(
  double t,
  std::tuple<double *, vec<3> *, vec<3> *>,
  unsigned int n
);
```

1. If the t-value is one of the $n+1$ entries of the array `ts` (the first item in the tuple), then you will return the corresponding item in the array `ys` (the second item in the tuple).
2. If the t-value is outside the interval $[t_0, t_f]$, you will throw a `std::domain_error` with an appropriate error message.
3. Otherwise, the point t will be between two values $t_{k-1}, t_k)$ for some value of $k = 1, \ldots, n$. You will then find three cubic splines and evaluate them at the point t. For this, you will have to pass t and the appropriate three pairs of values as two-dimensional arrays.

Here is the source code for the cubic spline algorithm as presented in class. You can either use this function three times, or if you are clever, you will be able to change the types of some of the parameters so that the return type is `vec<3>` and where you just pass `t, get<0>( y ) + k, get<1>( y ) + k, get<2>( y ) + k )` for an appropriate value of `k`. Recall that if `ptr` is a pointer storing the address of an array, then `ptr + k` is the address of `ptr[k]`, and you can treat `ptr + k` as an array starting at `ptr[k]`.

If you think smart and change `ivp_spline_2pt`, then this is really simple.

```
double ivp_spline_2pt(
    double  t,
    double  ts[2],
    double  ys[2],
    double dys[2]
) {
    double     h{ ts[1] -ts[0] };
    double delta{   (t - ts[0])/h };
    assert( (0.0 <= delta) && (delta <= 1.0) );

    return (
        (
            (
                h*(dys[0] + dys[1]) + 2.0*(ys[0] -ys[1])
            )*delta - (h*(2.0*dys[0] + dys[1]) + 3.0*(ys[0] -ys[1]))
        )*delta + h*dys[0]
    )*delta + ys[0];
}
```

6 Printing the output

The following code will print the output in a standardized fashion:

```
void print( std::tuple<double *, vec<3> *, vec<3> *> out, unsigned int n ) {
    std::cout << "The " << (n + 1) << " t-values:" << std::endl;

    std::cout << std::get<0>( out )[0];

    for ( std::size_t k{ 1 }; k <= n; ++k ) {
        std::cout << ", " << std::get<0>( out )[k];
    }

    std::cout << std::endl << std::endl;

    for ( std::size_t i{ 0 }; i < 3; ++i ) {
        std::cout << "The approximation of y" << i << "(t[k]):" << std::endl;

        std::cout << std::get<1>( out )[0]( i );

        for ( std::size_t k{ 1 }; k <= n; ++k ) {
            std::cout << ", " << std::get<1>( out )[k]( i );
        }

        std::cout << std::endl;
    }

    std::cout << std::endl;

    for ( std::size_t i{ 0 }; i < 3; ++i ) {
        std::cout << "The approximation of y" << i << "'(t[k]):" << std::endl;

        std::cout << std::get<2>( out )[0]( i );

        for ( std::size_t k{ 1 }; k <= n; ++k ) {
            std::cout << ", " << std::get<2>( out )[k]( i );
        }

        std::cout << std::endl;
    }

    std::cout << std::endl;
}
```

7 Testing your code

If you use the following right-hand side function of the derivative,

```
vec<3> f2( double t, vec<3> y ) {
  return vec<3>{
    -0.3*y(0) + 0.09*y(1) - 0.11*y(2),
    -0.7*y(1) - 0.08*y(0) + 0.03*y(2),
    -0.4*y(2) + 0.02*y(0) - 0.15*y(1)
  };
}
```

Then the output of

```
int main() {
  std::cout.precision( 6 );
  auto result{ rk4(
    f2, std::make_pair( 0.0, 4.0 ), {1.0, 2.0, 3.0}, 8
  ) };

  print( result, 8 );

  delete[] std::get<0>( result );
  delete[] std::get<1>( result );
  delete[] std::get<2>( result );

  return 0;
}
```

should be

```
The 9 t-values:
0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4

The approximation of the three functions:
1, 0.795519, 0.627981, 0.49167, 0.3815, 0.293036, 0.222463, 0.166541, 0.122541
2, 1.4129, 0.998206, 0.705499, 0.499039, 0.353501, 0.250957, 0.178726, 0.127844
3, 2.35013, 1.8499, 1.46263, 1.16114, 0.925177, 0.739568, 0.592879, 0.476447

The slopes at the approximations:
-0.45, -0.370009, -0.302044, -0.244895, -0.197262, -0.157865, -0.125505, -0.0990936, -0.0776655
-1.39, -0.982169, -0.693486, -0.489304, -0.345013, -0.243138, -0.17128, -0.120645, -0.0850007
-1.48, -1.13608, -0.87713, -0.681042, -0.531681, -0.417235, -0.329021, -0.26063, -0.207304
```

8 Discussion (which you may skip)

A C++ class must occupy a fixed amount of memory that can be calculated at compile time. This is because the compiler must know how much memory must be requested if, for example, you create an array of a class. Thus, there are two approaches to creating a vector class for linear algebra:

1. Have each vector store its dimension (capacity) and then dynamically allocate an array of that given dimension.
2. Have a templated class such as `vec<N>` where the user must explicitly specify the dimension in the source code, in which case, there is no requirement for dynamically allocated memory. Instead, each instance of the class has an array of the given capacity, and there isn't even a requirement to store the dimension of the array.

In the first case, there is much more flexibility: you can create an array at run time of whatever dimension you want. However, in this case, each time you perform any operation between to vectors or a matrix and a vector, you must explicitly check that the dimensions correctly match. In the second case, because all the dimensions must be specified in the source code, the compiler can check that the dimensions are correct, so if you tried the following:

```
vec<3> u{ 1.2, 3.5, 4.3 };
vec<4> v{ 7.2, 4.5, 9.2, 8.1 };
std::cout << (u + v) << std::endl;
```

then the compiler would signal an error: there are no algorithms for adding vectors of different dimensions, and nor do you want such an algorithm. Consequently, the execution of algorithms is always faster, and the memory requirements are lower: there is no need to store either a pointer nor the capacity.

9 Runge-Kutta is $O(h^4)$

You will show that the 4th-order Runge Kutta method is $O(h^4)$ as follows:

Suppose we have an initial-value problem that contains system of linear ordinary differential equations (LODE), for example,

$$\begin{aligned} y_1^{(1)}(t) &= -0.49y_1(t) + 0.89y_2(t) - 0.03y_3(t) \\ y_2^{(1)}(t) &= -0.79y_1(t) - 0.43y_2(t) - 0.45y_3(t) \\ y_3^{(1)}(t) &= 0.03y_1(t) + 0.56y_2(t) - 0.38y_3(t) \\ y_1(0) = 1,\ & y_2(0) = 2, y_3(0) = 3 \end{aligned} \tag{2}$$

You will notice that because it is linear, so we can write this as a matrix-vector equation, so let

$$\mathbf{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{pmatrix},$$

and therefore

$$\mathbf{y}^{(1)}(t) = \begin{pmatrix} y_1^{(1)}(t) \\ y_2^{(1)}(t) \\ y_3^{(1)}(t) \end{pmatrix},$$

and thus we may write this system of LODEs as

$$\mathbf{y}^{(1)}(t) = A\mathbf{y}(t)$$

where

$$A = \begin{pmatrix} -0.49 & 0.89 & -0.03 \\ -0.79 & -0.43 & -0.45 \\ 0.03 & 0.56 & -0.38 \end{pmatrix}.$$

In Matlab, you can create this matrix and a vector of the initial values:

```
A = [-0.49    0.89   -0.03
    -0.79   -0.43   -0.45
     0.03    0.56   -0.38]
y0 = [1 2 3]';
```

You can now find the eigenvectors and eigenvalues of this matrix:

```
[E, D] = eigs( A )
   E =
      0.03973 - 0.61930i   0.03973 + 0.61930i  -0.49454 + 0.00000i
      0.67988 + 0.00000i   0.67988 - 0.00000i  -0.01645 + 0.00000i
     -0.04532 - 0.38807i  -0.04532 + 0.38807i   0.86900 + 0.00000i

   D =
     -0.44616 + 0.97645i                    0                    0
                       0  -0.44616 - 0.97645i                    0
                       0                    0  -0.40767 + 0.00000i
```

You will recall from your course in linear algebra that most matrices A (and all matrices that have non-repeated eigenvalues) can be written in the form $A = EDE^{-1}$, and if A is symmetric (self-adjoint), then E can be made into an orthogonal matrix (all columns are normalized and all columns are mutually orthogonal, thus forming an orthonormal bases), and if E is an orthogonal matrix, then $E^{-1} = E^{\top}$. Unfortunately, in this case, A is not symmetric, and therefore E^{-1} cannot be easily calculated. Thus, we will take steps to avoid calculating E^{-1} in the subsequent calculations.

Beyond the scope of this course, the solution to the initial-value problem is $\mathbf{y}(t) = Ee^{tD}E^{-1}\mathbf{y}_0$ where

$$e^M = \sum_{k=0}^{\infty} \frac{M^k}{k!} = Id + M + \frac{M^2}{2} + \frac{M^3}{3!} + \frac{M^4}{4!} + \cdots$$

where Id is the identity matrix. In general, this is very difficult to calculate, but if the matrix is a diagonal matrix, then

$$e^D = e^{\begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}} = \begin{pmatrix} e^{\lambda_1} & & \\ & \ddots & \\ & & e^{\lambda_n} \end{pmatrix}.$$

Thus, we have:

```
y = @(t)( real( E*expm(t*D)*(E\y0) ) );
```

You will notice that rather than calculating $E^{-1}\mathbf{y}_0$, we use Matlab's solver. Also, because the eigenvalues are complex, the resulting vector will have small imaginary values (on the order of $10^{-16}j$). We don't need these values, so we will eliminate them, as all these functions must be real-valued functions of a real variable.

Note that `exp(A)` in Matlab replaces each entry with e raised to that power, so to calculate the matrix exponential, you must use the function `expm(A)`.

In Matlab, you will calculate $\mathbf{y}(10)$ and copy this back into your code in C++. You will then find the solution with

```
  vec<3> g( double t, vec<3> y ) {
    return vec<3>{
      -0.49*y(0) + 0.89*y(1) - 0.03*y(2),
      -0.79*y(0) - 0.43*y(1) - 0.45*y(2),
       0.03*y(0) + 0.56*y(1) - 0.38*y(2)
    };
  }
```

approximate a solution using $n = 2, 4, 8, 16, 32$ and 64 to approximate $\mathbf{y}(10)$.

10 Generalizing to n dimensions (which you may skip)

You can generalize your code to solving an initial-value problem of n coupled differential equations with one small trick: Make your function a templated function where you have the template variable be N:

```
template <unsigned int N>
std::tuple<double *, vec<N> *, vec<N> *> rk4(
  std::function<vec<N>( double t, vec<N> y )> f,
  std::pair<double, double> t_rng, vec<N> y0,
  unsigned int n
);
```

and then everywhere in the function you use `vec<3>`, replace that with `vec<N>`.

11 The Lorenz equations (which you may skip)

The Lorenz equations, when plotted as points in $\mathbf{R}^3$, form the *butterfly* image shown at `https://en.wikipedia.org/wiki/Lorenz_system` and what we do is we calculate $\mathbf{y}(t)$ for many values of t and then plot the outputs in $\mathbf{R}^3$. In the image on Wikipedia, you can see how $\mathbf{y}(t)$ changes over time. You can recreate this by using the function `lorenz` with your 4th-order Runge Kutta function.