DISCLAIMER

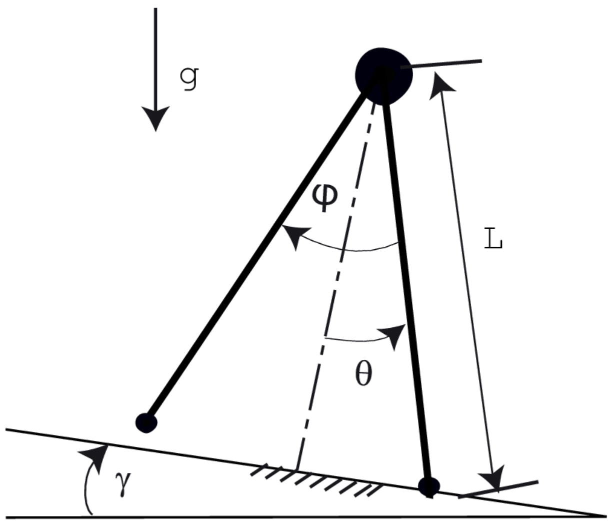
The *preliminary Spec Tests* that you are able to run only test that your code compiles and provides an indication about how well your code can be tested with the *complete Spec Test*. The *preliminary Spec Tests* **DO NOT TEST** the correct functioning of your code.
The maximum score you can obtain with the *preliminary Spec Test* is `10 / 100`. If you obtain this maximum score it indicates that your code is likely compatible with the *complete Spec Test*.
It is your responsibility to test that your code is correct.

After the deadline, your code will be tested with the *complete Spec Test* which does test the correct functioning of your code.
The maximum score you can obtain with the *complete Spec Test* is `100 / 100`.
The *complete Spec Test* will determine your grade for this assignment.

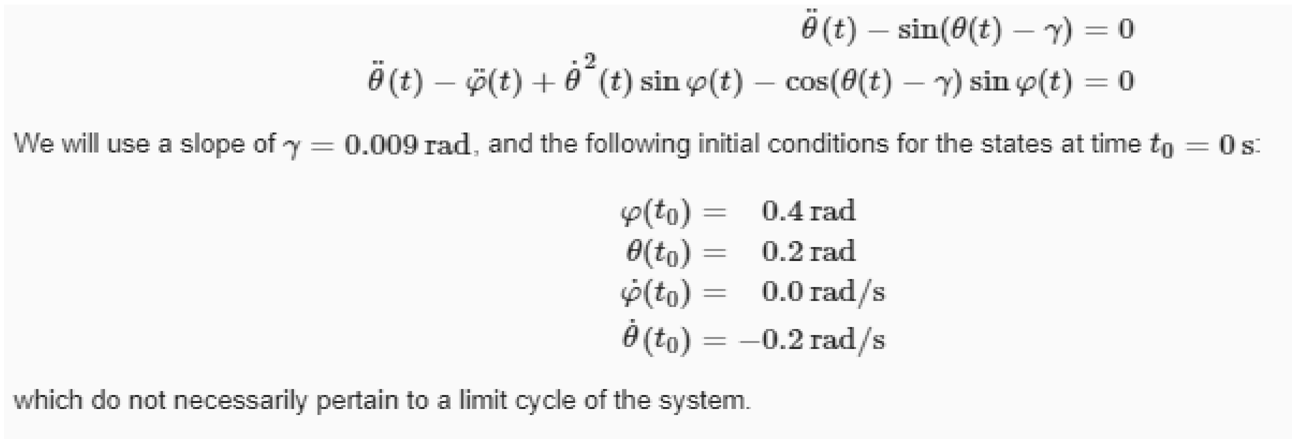
General problem description

You will write C++ code to solve an interesting dynamics problem. Walking robots have gained interest in recent years. While you can find complex and sophisticated walking robots online, today, you will consider the so-called **simplest walker**, as described by Garcia et al. in https://doi.org/10.1115/1.2798313 (available as PDF).

The simplest walker has two rigid massless legs that are hinged at the hip, and point-masses at the feet. The mass of these feet is negligible compared to a point mass at the hip. It walks down a slope with constant inclination γ. We assume that the stance leg does not slip nor rebound, such that the foot-ground contact may be modeled as a hinge joint as well. The definition of the stance leg angle φ and swing leg angle θ are shown in the figure below



Given are simplified second-order differential equations of motion of the simplest walker when it is supported on one leg (which is called single support), in the generalised coordinates θ and φ:

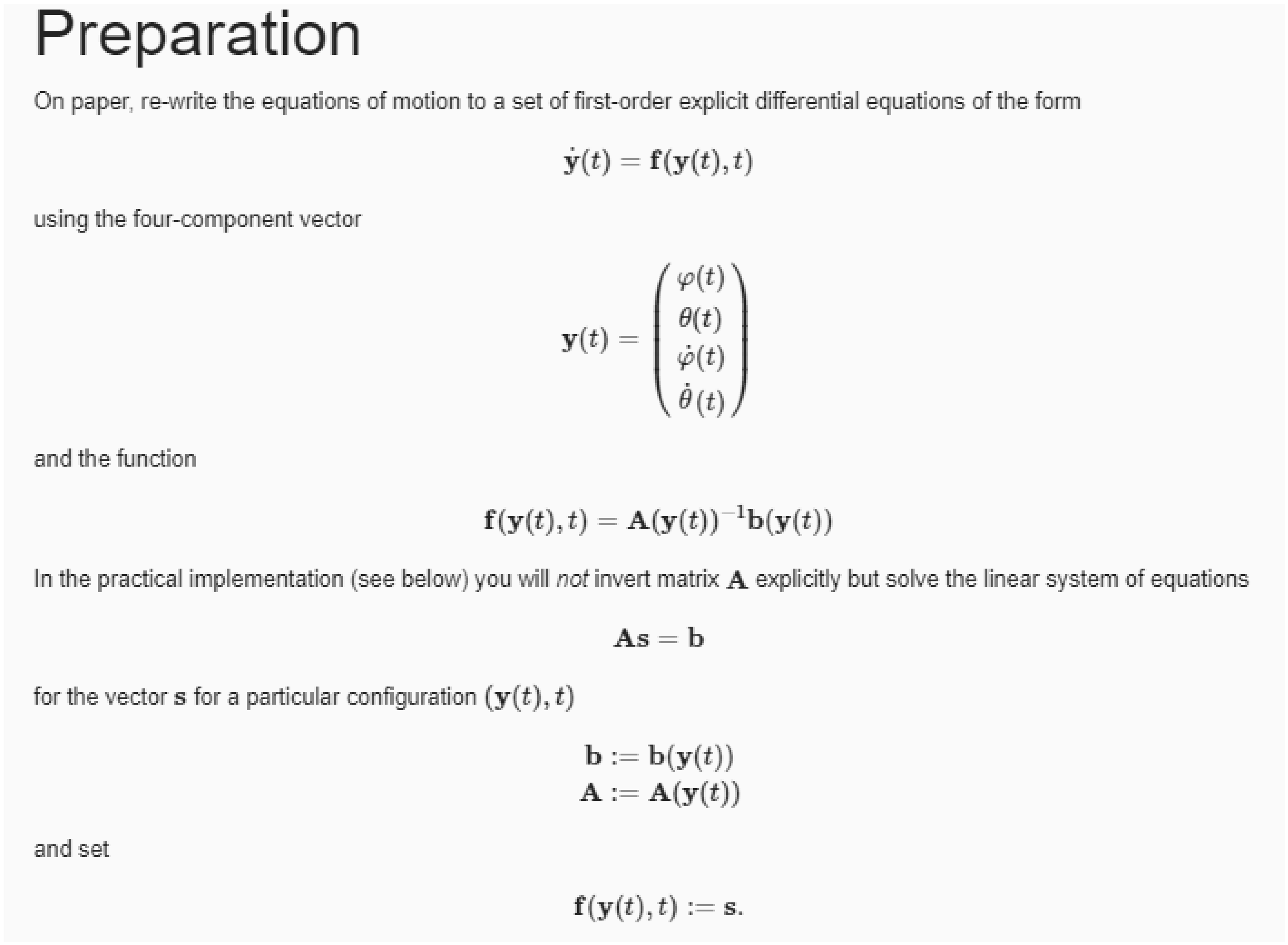


$$\ddot{\theta}(t) - \sin(\theta(t) - \gamma) = 0$$
$$\ddot{\theta}(t) - \ddot{\varphi}(t) + \dot{\theta}^2(t)\sin\varphi(t) - \cos(\theta(t) - \gamma)\sin\varphi(t) = 0$$

We will use a slope of $\gamma = 0.009\,\mathrm{rad}$, and the following initial conditions for the states at time $t_0 = 0\,\mathrm{s}$:

$$\begin{aligned} \varphi(t_0) &= \quad 0.4\,\mathrm{rad} \\ \theta(t_0) &= \quad 0.2\,\mathrm{rad} \\ \dot{\varphi}(t_0) &= \quad 0.0\,\mathrm{rad/s} \\ \dot{\theta}(t_0) &= -0.2\,\mathrm{rad/s} \end{aligned}$$

which do not necessarily pertain to a limit cycle of the system.



Preparation

On paper, re-write the equations of motion to a set of first-order explicit differential equations of the form

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

using the four-component vector

$$\mathbf{y}(t) = \begin{pmatrix} \varphi(t) \\ \theta(t) \\ \dot{\varphi}(t) \\ \dot{\theta}(t) \end{pmatrix}$$

and the function

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

In the practical implementation (see below) you will *not* invert matrix $\mathbf{A}$ explicitly but solve the linear system of equations

$$\mathbf{A}\mathbf{s} = \mathbf{b}$$

for the vector $\mathbf{s}$ for a particular configuration $(\mathbf{y}(t), t)$

$$\begin{aligned} \mathbf{b} &:= \mathbf{b}(\mathbf{y}(t)) \\ \mathbf{A} &:= \mathbf{A}(\mathbf{y}(t)) \end{aligned}$$

and set

$$\mathbf{f}(\mathbf{y}(t), t) := \mathbf{s}.$$

Implementation

First, you will implement the basic building blocks, that is, `Vector` and `Matrix` classes, the `bicgstab` function (for solving linear systems of equations) and `heun` function (for computing the solution to explicit differential equations).

Next, you will implement the `SimplestWalker` class, which sets up the problem in the constructor and provides a `step` function to advance the walker in time.

Finally, you will implement the `main` function that tests your model for a given problem configuration.

Note: All integer variables should have type `int`. Do not use `std::size_t` or other types since this might lead to problems with our Spec Tests.

Vector

1. Create the class

```
template <typename T>
class Vector
{...};
```

whereby the `Vector`'s elements are of type `T`. The `Vector` class must provide the following functionality:

Constructors and destructor

- A **default constructor** that sets the length to zero.
- A **copy constructor** and a **move constructor** that creates a `Vector` from another `Vector`.
- A **constructor** that takes a length as an argument and allocates the internal data structures accordingly.
- A **constructor** that takes an initialiser list representing the contents of this `Vector`, allocates the internal data structures and initialises the `Vector`'s content accordingly.
- A **destructor**.

Operators and functions

- A **copy assignment operator** and a **move assignment operator** from another `Vector`.
- An `operator[](int i)` that returns a reference to the `i`-th entry of the vector. Implement an overload of this operator that returns a *constant* reference. Both operators can be used to access the entries in functions that are implemented outside the `Vector` class.
- **Arithmetic operators** `operator+` and `operator-` to add and subtract two `Vector`s. These operators must support `Vector`s of different types, whereby the resulting `Vector` has to be of the type that dominates (e.g., `double` dominates `float`). If the `Vector`s have different lengths, all operators must throw an exception.
- **Arithmetic operators** `operator*` between a scalar and a `Vector` ($\mathbf{w} = s \cdot \mathbf{v}$) and a `Vector` and a scalar ($\mathbf{w} = \mathbf{v} \cdot s$), whereby the scalar and the `Vector` can have different types and the resulting `Vector` must be of the dominating type.
- A function `len` that returns the length of the `Vector`. This function can be used to retrieve the length in functions that are implemented outside the `Vector` class.

2. Create a function `dot` that computes the standard inner product of two `Vector` s. The function must have the following signature:

   ```cpp
   template<typename T, typename U>
   typename std::common_type<T,U>::type
   dot(const Vector<T>& lhs,
       const Vector<U>& rhs)
   {...}
   ```

   If the `Vector` s have different lengths, the `dot` function must throw an exception.

3. Create a function `norm` that computes the l^2-norm of a `Vector` . The function must have the following signature:

   ```cpp
   template<typename T>
   T
   norm(const Vector<T>& vec)
   {...}
   ```

Matrix

1. Create the class

```
template <typename T>
class Matrix
{...};
```

that represents a sparse matrix, whereby the `Matrix`'s elements are of type `T`. A sparse matrix is a matrix in which most elements are zero so that it makes sense to store only the non-zero entries to save memory.

Consider for example the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 2 & 3 & 0 \\ 0 & 0 & 5 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

with only 7 (out of 16) non-zero entries. If we store the non-zero entries in the form

$$\begin{aligned} (0,0) &\to 1 \\ (0,3) &\to 1 \\ (1,1) &\to 2 \\ (1,2) &\to 3 \\ (2,2) &\to 5 \\ (3,0) &\to 1 \\ (3,3) &\to 1 \end{aligned}$$

then the matrix-vector multiplication

$$\mathbf{y} = \mathbf{A}\,\mathbf{x}$$

can be realised in a loop over all non-zero entries $\mathbf{ij}$ of the matrix $\mathbf{A}$ with the update formula

$$\mathbf{y_i} := \mathbf{y_i} + \mathbf{A_{ij}}\,\mathbf{x_j}$$

Hint: Use the `std::map` class from the C++ standard library to store the non-zero matrix entries. Use `std::pair<int, int>` as the key type to store the row and column positions and `T` as the data type to store the value at the particular matrix position.

Even though `std::map` allows to *search* for a key it is not a good idea to use this feature for the sparse matrix-vector multiplication since this is very inefficient. Instead, use an iterator over all entries of the map and extract row and column positions and the data value via

```
int i   = it->first.first;
int j   = it>-first.second;
T value = it->second;
```

The `Matrix` class must provide the following functionality:

- A **constructor** that accepts two integer values (number of rows and columns) and initialises the internal data structures.

  ```
  Matrix<double> M(10, 20); // initialise M with 10 rows and 20 columns
  ```

  It is not allowed to change the dimensions of the matrix afterwards.

- A **destructor**.

- An `operator[](const std::pair<int, int>& ij)` that returns the matrix entry `ij` (i.e. the entry at row i and column j) by reference. If that entry is not yet present, the operator should create it.

  ```
  M[{0,0}] = 1.0; // set value at row 0, column 0 to 1.0
  M[{1,2}] = 2.0; // set value at row 1, column 2 to 2.0
  ```

- An `operator()(const std::pair<int, int>& ij) const` that returns the matrix entry `ij` by *constant* reference and throws an exception if the entry is not present.

  ```
  std::cout << M({0,0}) << std::endl; // prints 1.0
  std::cout << M({3,3}) << std::endl; // throws an exception
  ```

- An `operator*` between a `Matrix` and a `Vector` object that implements the sparse matrix-vector product. The operator must have the following signature:

  ```
  template<typename T, typename U>
  Vector<typename std::common_type<T,U>::type>
  operator*(const Matrix<T>& lhs,
            const Vector<U>& rhs)
  { ... }
  ```

  If the dimension of the `Matrix` and the `Vector` are not compatible, the operator must throw an exception.

Biconjugate stabilized gradient method

1. Create a function named `bicgstab` that solves a linear system of equations using the Biconjugate Stabilized Gradient method (BiCGStab), given in pseudocode by

```
q_0 = r_0 = b - A * x_0
v_0 = p_0 = 0
alpha = rho_0 = omega_0 = 1

for k = 1, 2, ..., maxiter
    rho_k = dot(q_0, r_(k-1))
    beta  = (rho_k / rho_(k-1)) * (alpha / omega_(k-1))
    p_k   = r_(k-1) + beta (p_(k-1) - omega_(k-1) * v_(k-1))
    v_k   = A * p_k
    alpha = rho_k / dot(q_0, v_k)
    h     = x_(k-1) + alpha * p_k

    if norm(b - A * h) < tol
       x_k = h
       stop

    s       = r_(k-1) - alpha * v_k
    t       = A * s
    omega_k = dot(t, s) / dot(t, t)
    x_k     = h + omega_k * s

    if norm(b - A * x_k) < tol
       stop

    r_k = s - omega_k * t
```

Here, `A` is a `Matrix` object, `x_k`, `p_k`, `q_0`, `p_k`, `v_k`, `h`, `s` and `t` are `Vector` objects, `dot` is the standard l^2-inner product, `norm` is the standard l^2-norm, `tol` is an absolute tolerance for the residual and `maxiter` is the maximum allowed number of iterations.

The `bicgstab` function must have the following signature:

```
template<typename T>
int bicgstab(const Matrix<T> &A,
             const Vector<T> &b,
             Vector<T>       &x,
             T                tol     = (T)1e-8,
             int              maxiter = 100)
{ ... }
```

The third argument serves both as the initial guess (`x_0` in the pseudocode) and as the result (`x_k` in the pseudocode, where `k` is the last iteration). The function must return the number of iterations used to achieve the desired tolerance `tol` if the BiCGStab method converged within `maxiter` iterations and `-1` otherwise, that is, if `maxiter` iterations have been reached without reaching convergence.

Heun’s integration method

1. Create a function named `heun` that solves a system of first-order explicit ordinary differential equations by Heun’s method also called modified Euler method from time t_n to t_{n+1} via

$$\begin{aligned}\bar{\mathbf{y}}_{\mathbf{n+1}} &= \mathbf{y_n} + h \cdot \mathbf{f}(\mathbf{y_n}, t_n)\\ \mathbf{y}_{\mathbf{n+1}} &= \mathbf{y_n} + \frac{h}{2} \cdot \left(\mathbf{f}(\mathbf{y_n}, t_n) + \mathbf{f}(\bar{\mathbf{y}}_{\mathbf{n+1}}, t_{n+1})\right)\end{aligned}$$

   Before you implement it, make a drawing for yourself to explain how this method approximates the derivative.

   The `heun` function must have the following signature:

```
template<typename T>
void heun(const Vector<std::function<T(const Vector<T>&, T)> >& f,
                Vector<T>&                                      y,
                T                                               h,
                T&                                              t)
{ ... }
```

   - The first argument `const Vector<std::function<T(const Vector<T>&, T)> > f` is used to pass the vector-valued function $\mathbf{f}(\mathbf{y}(t), t)$ as a vector of `std::function` ’s.

     As an illustration how this function can be implemented, consider the following vector-valued function, which is not part of the bipedal robot model and only serves for illustration purposes:

$$\mathbf{y}(t) = \begin{pmatrix} \phi_1(t)\\ \phi_2(t)\\ \dot{\phi}_1(t)\\ \dot{\phi}_2(t)\end{pmatrix}, \quad \mathbf{f}\left(\mathbf{y}(t), t\right) = \begin{pmatrix} 2t\dot{\phi}_1(t)\\ 3t\dot{\phi}_2(t)\\ t\phi_1(t)\\ 2t\phi_2(t)\end{pmatrix}$$

```
Vector<std::function<double(const Vector<double>&, double)> > f =
{
[](Vector<double> const& y, double t) { return 2 * t * y[2]; },
[](Vector<double> const& y, double t) { return 3 * t * y[3]; },
[](Vector<double> const& y, double t) { return     t * y[0]; },
[](Vector<double> const& y, double t) { return 2 * t * y[1]; },
};
```

     Each single entry $\mathbf{f_i}(\mathbf{y}(t), t)$ is of the form `std::function<T(const Vector<T>&, T)>` , that is, it is a function that accepts the state vector $\mathbf{y}(t)$ as first argument `Vector<T>` (passed by constant reference) and time t as second argument and implements the actual function as a lambda expression.

   - The second argument `Vector<T>& y` to the `heun` function serves both as input ($\mathbf{y_n}$) and as the result ($\mathbf{y}_{\mathbf{n+1}}$). The third and fourth arguments, `T h` and `T &t` , denote the step size h and the time level t_n, respectively. The function should return the next time level $t_{n+1} = t_n + h$ via the parameter `t` .

     Hint: Note that for the simplest walker $\mathbf{f}(\mathbf{y}(t), t) = \mathbf{s}(t)$, where $\mathbf{s}(t)$ is the solution of the linear system of equations $\mathbf{A}(t)\mathbf{s}(t) = \mathbf{b}(t)$ (see “General problem description” above). Think about possibilities to provide this information, e.g., as attributes of the `SimplestWalker` class (see below). Keep in mind that matrix $\mathbf{A}$ and/or the right-hand side vector $\mathbf{b}$ may depend on $\mathbf{y}(t)$ and need to be updated between the two steps of Heun’s method.

Simplest Walker

1. Create the class

```cpp
template <typename T>
class SimplestWalker
{...};
```

that must provide the following functionality:

- A **constructor** that accepts the initial conditions ($\mathbf{y_0} = \mathbf{y}(t_0)$), the initial time ($t_0$), and the slope ($\gamma$) as input and stores them internally. The constructor must have the following signature:

```cpp
SimplestWalker(const Vector<T>& y0,
                      T          t0,
                      T          gamma)
```

- A function `derivative` that accepts vector

$$\mathbf{y} = \begin{pmatrix} \varphi \\ \theta \\ \dot{\varphi} \\ \dot{\theta} \end{pmatrix}$$

as input and produces its derivative $\dot{\mathbf{y}}$ as output. The `derivative` function must have the following signature:

```cpp
Vector<T> derivative(const Vector<T>& y) const
```

- A function `step` that accepts the time-step size h as input, performs a single time step by the `heun` function and updates $\mathbf{y_{n+1}}$ and t_{n+1}. The `step` function must return the updated state vector by constant reference and must have the following signature:

```cpp
const Vector<T>& step(T h)
```

Testing your implementation

1. Test your implementation of the simplest walker for the initial conditions for the states at time $t_0 = 0\,\mathrm{s}$:

$$\begin{aligned} \varphi(t_0) &= 0.4\,\mathrm{rad} \\ \theta(t_0) &= 0.2\,\mathrm{rad} \\ \ddot{\varphi}(t_0) &= 0.0\,\mathrm{rad/s} \\ \ddot{\theta}(t_0) &= -0.2\,\mathrm{rad/s}, \end{aligned}$$

and a slope of $\gamma = 0.009\,\mathrm{rad}$ for a duration of 2 seconds with time step size $h = 0.001\,\mathrm{s}$.