Exploring Physics with C++

Zvonimir Vanjak

EXPLORING PHYSICS WITH C++

Contents

Preface

Motivacija

- Iako ima knjiga koje koriste C++ kao programski jezik za prezentaciju tehnika numerical computinga, taj C++ je zastarjelog (C-like) stila I ne koristi ni blizu sve mogućnosti modernog C++a
- Što se tiče korištenja softvera za istraživanje fizike, postoji priličpno opsežna lioteratura u kojoj se koriste Matlab I Mathematica, u zadnjih par godina ima I sve više knjiga koje koriste Python u takvu svrhu, ali nema baš takve knjige za C++

Osobno

- Tema me fascinira 30 godina, otkad sam prvi put u ruke uzeo Numerical Recipes in C
- A fascinira me I fizika ❤️‍🩹
- I izvrsna je prilika za re-learning, I fizike I ažuriranje znanja C++a

Osnovni cilj je istražiti različite fizikalne pojave, zakonitosti I generalno različite situacije, koristeći C++ za provođenje numeričkih proračuna

Te uz to opisati I relevantnu fiziku I osnovne tehnike numerical computinga koje će se koristiti

Nekakav početni okvir za raspored po količini sadržaja bi bio: 30% fizike, 40% numerical computing, 30% konkretan C++ kôd

Pregled poglavlja

1. Prva tri – osnovni objekti, algoritmi I vizualizacija
2. Osnovna mehanika – 4, 5, 6, I 7
3. Gravity – 8 I 9
4. Inertial, non-inertial frames and coordinate transformations – 10, 11 I 12
5. Rigid body – 13 I 14
6. Special relativity – 15 I 16
7. Electromagnetism – 17, 18, 19
8. Curves and surfaces – 20
9. General relativity - 21

Acknowledgments

Mater i ćaća

Obitelj

Stric Andrija

Profesori Jamničić i Simić

FER - Kalpić i Mornar

Zahvale contributorima

1. Essential mathematical objects

Where we start from the beginning.

PRELIMINARIES

Doing numerics on computer – challenges, pitfalls and traps

Representing numbers

Precision of real representation

about IEEE754, mantissa, exponents, and all that

TODO – table with precision and ranges for single and double precision

Roundoff errors

Couple of examples of roundoff errors

Special emphasis on subtraction of similar numbers

Truncation errors

Numerical computing in modern C++

TODO – minimal history, advances, still going extra strong, future plans for C++26

Has come a long way since times when even PI was not part of the standard.

Still, there is no POW2 in C++!

But there are great numerical libraries

GSL – GNU Scientific Library

Link and short overview.

Boost.Math

Link, overview.

Especially Boost.Math toolkit.

Overview of other C++ numerical libraries of note

Starting our own Minimal Math Library

Why not just use Boost or GSL?

- Learning by implementing
- Creating general math library that is versatile and easily usable
- And that also has basic mathematical objects directly modelled (in today's parlance, you could say it is "opinionated")

C++20 is our starting point

MMLBase.h

This will be the base header for our library.

Starting with necessary headers we need to include

```
#include <stdexcept>
#include <initializer_list>
#include <algorithm>
#include <memory>
#include <functional>

#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <iomanip>

#include <cmath>
#include <limits>
#include <complex>
#include <numbers>
```

Following good programming practice, all our code will be within namespace MML

Some basics

```
namespace MML
{
  template<class Type>
  static Real Abs(const Type& a)
  {
    return std::abs(a);
  }
  template<class Type>
  static Real Abs(const std::complex<Type>& a)
  {
    return hypot(a.real(), a.imag());
  }
  inline bool isWithinAbsPrec(Real a, Real b, Real eps)
  {
    return std::abs(a - b) < eps;
  }
  inline bool isWithinRelPrec(Real a, Real b, Real eps)
  {
    return std::abs(a - b) < eps * std::max(Abs(a), Abs(b));
  }

  template<class T> inline T POW2(const T a) { const T t=a; return t * t; }
  template<class T> inline T POW3(const T a) { const T t=a; return t * t * t; }
  template<class T> inline T POW4(const T a) { const T t=a; return t * t * t * t; }
```

Constants

TODO – do some thinking about this (and possibly expand list!)

```
////////////                  Constants                ////////////////
namespace Constants
{
  static inline const Real PI = std::numbers::pi;
  static inline const Real Epsilon = std::numeric_limits<Real>::epsilon();
  static inline const Real PositiveInf = std::numeric_limits<Real>::max();
  static inline const Real NegativeInf = -std::numeric_limits<Real>::max();
}
```

Some defaults for precision of numerical operations

```
namespace Defaults
{
  static int VectorPrintWidth = 15;
  static int VectorPrintPrecision = 10;

  //////////              Default precisions             ///////////
  // TODO - how to make dependent on Real type
  // //     (ie. different values for float, double and long double)
  static inline const double ComplexEqualityPrecision = 1e-15;
  static inline const double ComplexAbsEqualityPrecision = 1e-15;
  static inline const double MatrixEqualityPrecision = 1e-15;
  static inline const double VectorEqualityPrecision = 1e-15;

  static inline const double IsMatrixSymmetricPrecision = 1e-15;
  static inline const double IsMatrixDiagonalPrecision = 1e-15;
  static inline const double IsMatrixUnitPrecision = 1e-15;
  static inline const double IsMatrixOrthogonalPrecision = 1e-15;
```

And list of all available standard and special functions in C++20 standard

```
static inline Real Sin(Real x) { return sin(x); }
static inline Real Cos(Real x) { return cos(x); }
static inline Real Sec(Real x) { return 1.0 / cos(x); }
static inline Real Csc(Real x) { return 1.0 / sin(x); }
static inline Real Tan(Real x) { return tan(x); }
static inline Real Ctg(Real x) { return 1.0 / tan(x); }

static inline Real Exp(Real x) { return exp(x); }
static inline Real Log(Real x) { return log(x); }
static inline Real Log10(Real x){ return log10(x); }
static inline Real Sqrt(Real x) { return sqrt(x); }
static inline Real Pow(Real x, Real y) { return pow(x, y); }

static inline Real Sinh(Real x) { return sinh(x); }
static inline Real Cosh(Real x) { return cosh(x); }
static inline Real Sech(Real x) { return 1.0 / cosh(x); }
static inline Real Csch(Real x) { return 1.0 / sinh(x); }
static inline Real Tanh(Real x) { return tanh(x); }
static inline Real Ctgh(Real x) { return 1.0 / tanh(x); }

static inline Real Asin(Real x) { return asin(x); }
static inline Real Acos(Real x) { return acos(x); }
static inline Real Atan(Real x) { return atan(x); }

static inline Real Asinh(Real x) { return asinh(x); }
static inline Real Acosh(Real x) { return acosh(x); }
static inline Real Atanh(Real x) { return atanh(x); }
```

Special functions

```
static inline Real Erf(Real x)  { return std::erf(x); }
static inline Real Erfc(Real x) { return std::erfc(x); }

static inline Real TGamma(Real x) { return std::tgamma(x); }
static inline Real LGamma(Real x) { return std::lgamma(x); }
static inline Real RiemannZeta(Real x) { return std::riemann_zeta(x); }
static inline Real Comp_ellint_1(Real x) { return std::comp_ellint_1(x); }
static inline Real Comp_ellint_2(Real x) { return std::comp_ellint_2(x); }

static inline Real Hermite(unsigned int n, Real x) { return std::hermite(n, x); }
static inline Real Legendre(unsigned int n, Real x) { return std::legendre(n, x); }
static inline Real Laguerre(unsigned int n, Real x) { return std::laguerre(n, x); }
static inline Real SphBessel(unsigned int n, Real x) { return std::sph_bessel(n, x); }
static inline Real SphLegendre(int n1, int n2, Real x) { return std::sph_legendre(n1, n2, x); }
```

MMLExceptions.h

Even though having exceptions in our code exacts (quite) small runtime penalty, it is a small price for having dependable way of handling exceptional situations .

Set of classes defined in MMLExceptions.h

```
namespace MML
{
  //////////             Vector error exceptions            //////////
  class VectorInitializationError { ... };
  class VectorDimensionError { ... };
  class VectorAccessBoundsError { ... };

  //////////             Matrix error exceptions            //////////
  class MatrixAllocationError { ... };
  class MatrixAccessBoundsError { ... };
  class MatrixDimensionError { ... };
  class SingularMatrixError { ... };

  //////////             Integration exceptions             //////////
  class IntegrationTooManySteps { ... };

  //////////           Interpolation exceptions             //////////
  class RealFuncInterpInitError { ... };

  //////////             Tensor exceptions                 //////////
  class TensorCovarContravarNumError { ... };
  class TensorCovarContravarArithmeticError { ... };
  class TensorIndexError { ... };

  //////////           Root finding exceptions              //////////
  class RootFindingError { ... };

  //////////            ODE solver exceptions               //////////
  class ODESolverError { ... };
}
```

Running code and examples

All code available on Github (https://github.com/zvanjak/ExploringPhysicsWithCpp)

Using Visual studio code with standard extensions for C++ development

Working with different compilers – tested on MSVC, Clang, gcc

Tested on different platforms – Windows, Linux, Mac.

VECTOR – "THE WORKHORSE"

What is a vector?

Can represent almost anything, and in our physics investigations we'll use it to represent Cartesian and spherical vectors, momentum and angular momentum, …

but in essence, it is an array of numbers, real or complex, of some determined size and it is present in almost all numerical calculations.

Templated – so it can contain any kind of type

Two types.

- Runtime size
- Compile time size

Vector<Type> - runtime size

Represents (mathematical!) vector of dynamically defined size, and it is basically a wrapper around std::vector<>.

Delegation, not inheritance! As we will be at the same time restricting set of operations available from std::vector<> and extending that set with some mathematical operations, not present in std::vector<>.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/Vector.h

```
template<class Type>
class Vector
{
private:
  std::vector<Type> _elems;

public:
  typedef Type value_type;      // make T available externally

  ////////////////////                Constructors                ////////////////////
  Vector() {}
  explicit Vector(int n) { ... }
  explicit Vector(int n, const Type &val) { ... }
  explicit Vector(int n, Type* vals) { ... }
  explicit Vector(std::vector<Type> values) : _elems(values) {}
  explicit Vector(std::initializer_list<Type> list) : _elems(list) {}
```

For accessing element, we implement two approaches – with and without bounds checking.

```
//////////////////////          Accessing elements            //////////////////////
inline Type&       operator[](int n)       { return _elems[n]; }
inline const Type& operator[](int n) const { return _elems[n]; }

// checked access
Type& at(int n) {
  if(n < 0 || n >= size())
    throw VectorDimensionError("Vector::at - index out of bounds", size(), n);
  else
    return _elems[n];
}
Type  at(int n) const {
  if(n < 0 || n >= size())
    throw VectorDimensionError("Vector::at - index out of bounds", size(), n);
  else
    return _elems[n];
}
```

Set of implemented arithmetic operators.

```
//////////////////////          Arithmetic operators          //////////////////////
Vector  operator-() const { ... }

Vector  operator+(const Vector& b) const { ... }
Vector& operator+=(const Vector& b) { ... }
Vector  operator-(const Vector& b) const { ... }
Vector& operator-=(const Vector& b) { ... }

Vector  operator*(Type b) { ... }
Vector& operator*=(Type b) { ... }
Vector  operator/(Type b) { ... }
Vector& operator/=(Type b) { ... }

friend Vector operator*(Type a, const Vector& b) { ... }
```

Operations for testing equality

```
//////////////////////            Testing equality            //////////////////////
bool operator==(const Vector& b) const { ... }
bool operator!=(const Vector& b) const { ... }
bool IsEqualTo(const Vector& b, Real eps = Defaults::VectorEqualityPrecision) const { ... }
bool IsNullVec() const { ... }
```

Calculating vector norm

```
//////////////////////                Operations
Real NormL1() const { ... }
Real NormL2() const { ... }
Real NormLInf() const { ... }
```

I/O for vectors

```
//////////////////////////             I/O              //////////////////////////
std::ostream& Print(std::ostream& stream, int width, int precision) const { ... }
std::ostream& Print(std::ostream& stream, int width, int precision, Real zeroThreshold) const { ... }
std::ostream& PrintLine(std::ostream& stream, const std::string &msg, int width, int precision) const { ... }

std::string to_string(int width, int precision) const { ... }
friend std::ostream& operator<<(std::ostream &stream, const Vector &a) { ... }
```

VectorN<Type, N> - compile-time size

Vector with statically defined size.

Needed mostly in its 2D, 3D or 4D incarnations, where it will be used as base class for some specific implementations.

Has mostly the same functionality as Vector, with obvious difference in data declaration and minor differences in constructors.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/VectorN.h

```
template<class Type, int N>  <T> Provide sample template arguments for IntelliSense
class VectorN
{
protected:
  Type  _val[N] = { 0 };

public:
  typedef Type value_type;      // make T available externally

  ////////////////////          Constructors and destructor       ////////////////////
  VectorN() {}
  explicit VectorN(const Type& init_val) { ... }
  explicit VectorN(std::initializer_list<Type> list) { ... }
  explicit VectorN(std::vector<Type> list) { ... }
  explicit VectorN(Type* vals) { ... }
```

Starting our BaseUtils class

Utility class with static helpers of various kinds and stuff that even though it is closely associated with Vector class, doesn't merit inclusion as pure member.

In some cases, it is actually impossible to preserve semantics, as in case of ScalarProduct() helper function where implementation for Real and Complex types is different.

Here are the vector helpers

```
//////////////////                  Vector helpers                  //////////////////
static bool AreEqual(const Vector<Real> &a, const Vector<Real> &b, Real eps = Defaults::VectorEqualityPrecision) { ... }
static bool AreEqual(const Vector<Complex>& a, const Vector<Complex>& b, double eps = Defaults::ComplexEqualityPrecision) { ...
static bool AreEqualAbs(const Vector<Complex>& a, const Vector<Complex>& b, double eps = Defaults::ComplexAbsEqualityPrecision)

static Real    ScalarProduct(const Vector<Real>& a, const Vector<Real>& b) { ... }
static Complex ScalarProduct(const Vector<Complex>& a, const Vector<Complex>& b) { ... }

template<int N>
static Real    ScalarProduct(const VectorN<Real, N> &a, const VectorN<Real, N> &b) { ... }
template<int N>
static Complex ScalarProduct(const VectorN<Complex, N>& a, const VectorN<Complex, N>& b) { ... }

static Vector<Real> VectorProjectionParallelTo(const Vector<Real>& orig, const Vector<Real>& b) { ... }
static Vector<Real> VectorProjectionPerpendicularTo(const Vector<Real>& orig, const Vector<Real>& b) { ... }

static Real VectorsAngle(const Vector<Real> &a, const Vector<Real> &b) { ... }
template<int N>
static Real VectorsAngle(const VectorN<Real, N> &a, const VectorN<Real, N> &b) { ... }
```

Unfortunately, mixing operations between different Vector types requires special functions.

```
//////////////////      Vector<Complex> - Vector<Real> operations      //////////////////
static Vector<Complex> AddVec(const Vector<Complex>& a, const Vector<Real>& b) { ... }
static Vector<Complex> AddVec(const Vector<Real>& a, const Vector<Complex>& b) { ... }

static Vector<Complex> SubVec(const Vector<Complex>& a, const Vector<Real>& b) { ... }
static Vector<Complex> SubVec(const Vector<Real>& a, const Vector<Complex>& b) { ... }
```

Example usage

TODO – example with vectors and its operations.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_01_basic_objects/demo_vector.cpp

MATRIX – "THE OMNIPRESENT"

Used everywhere …

Matrix<Type> - runtime size

Standard Matrix object with size defined at runtime, ie. at construction time.

Space for storing elements is allocated as contiguous block (not on a per row basis).

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/Matrix.h

Set of available constructors.

```
template<class Type>
class Matrix
{
private:
  int  _rows;
  int  _cols;
  Type** _data;

  void Init(int rows, int cols) { ... }

public:
  typedef Type value_type;      // make Type available externally

  ////////////////////          Constructors and destructor       ////////////////////
  explicit Matrix() : _rows(0), _cols(0), _data{ nullptr } {}
  explicit Matrix(int rows, int cols) { ... }
  explicit Matrix(int rows, int cols, const Type& val) { ... }
  // useful if you have a pointer to continuous 2D array (can be in row-, or column-wise memory layout)
  explicit Matrix(int rows, int cols, Type* val, bool isRowWise = true) { ... }
  // in strict mode, you must supply ALL necessary values for complete matrix initialization
  explicit Matrix(int rows, int cols, std::initializer_list<Type> values, bool strictMode = true) { ... }

  Matrix(const Matrix& m) { ... }
  // creating submatrix from given matrix 'm'
  Matrix(const Matrix& m, int ind_row, int ind_col, int row_num, int col_num) { ... }
  Matrix(Matrix&& m) { ... }
  ~Matrix() { ... }

  void Resize(int rows, int cols) { ... }
```

Basics stuff, and some manipulation routines.

```
//////////////////////              Standard stuff               //////////////////////
inline int RowNum() const { return _rows; }
inline int ColNum() const { return _cols; }

static Matrix GetUnitMatrix(int dim) { ... }
void   MakeUnitMatrix(void) { ... }

Matrix GetLower(bool includeDiagonal = true) const { ... }
Matrix GetUpper(bool includeDiagonal = true) const { ... }

void SwapRows(int k, int l) { ... }
void SwapCols(int k, int l) { ... }

//////////////////////         Matrix to Vector conversions      //////////////////////
Vector<Type> VectorFromRow(int rowInd) const { ... }
Vector<Type> VectorFromColumn(int colInd) const { ... }
Vector<Type> VectorFromDiagonal() const { ... }
```

Functions for calculating various matrix properties.

```
//////////////////////              Matrix properties            //////////////////////
bool IsUnit(double eps = Defaults::IsMatrixUnitPrecision) const { ... }
bool IsDiagonal(double eps = Defaults::IsMatrixDiagonalPrecision) const { ... }
bool IsDiagDominant() const { ... }
bool IsSymmetric() const { ... }
bool IsAntiSymmetric() const { ... }

Real NormL1() const { ... }
Real NormL2() const { ... }
Real NormLInf() const { ... }
```

Assignment and access operators.

```
//////////////////////            Assignment operators           //////////////////////
Matrix& operator=(const Matrix& m) { ... }
Matrix& operator=(Matrix&& m) { ... }

//////////////////////              Access operators             //////////////////////
inline Type*        operator[](int i)             { return _data[i]; }
inline const Type*  operator[](const int i) const { return _data[i]; }

inline Type  operator()(int i, int j) const { return _data[i][j]; }
inline Type& operator()(int i, int j)       { return _data[i][j]; }

// version with checked access
Type  at(int i, int j) const { ... }
Type& at(int i, int j) { ... }
```

Two ways to check for equality – exact, implemented with operators == and !=, and within given precision

```
//////////////////////              Equality operations          //////////////////////
bool operator==(const Matrix& b) const { ... }
bool operator!=(const Matrix& b) const { ... }

bool IsEqualTo(const Matrix<Type>& b, Type eps = Defaults::MatrixEqualityPrecision) const { ... }
static bool AreEqual(const Matrix& a, const Matrix& b, Type eps = Defaults::MatrixEqualityPrecision)
```

Standard set of arithmetic operators.

```
//////////////////////              Arithmetic operators          //////////////////////
Matrix  operator-() const { ... }

Matrix  operator+(const Matrix& b) const { ... }
Matrix& operator+=(const Matrix& b) { ... }
Matrix  operator-(const Matrix& b) const { ... }
Matrix& operator-=(const Matrix& b) { ... }
Matrix  operator*(const Matrix& b) const { ... }

Matrix  operator*(const Type &b) const { ... }
Matrix& operator*=(const Type &b) { ... }
Matrix  operator/(const Type &b) const { ... }
Matrix& operator/=(const Type& b) { ... }

Vector<Type> operator*(const Vector<Type>& b) const { ... }

friend Matrix operator*(const Type &a, const Matrix<Type>& b) { ... }
friend Vector<Type> operator*(const Vector<Type>& a, const Matrix<Type>& b) { ... }
```

Basic operations, where matrix inversion is performed by using Gauss-Jordan elimination with pivoting.

```
//////////////////////           Trace, Inverse & Transpose      //////////////////////
Type   Trace() const { ... }

void   Invert() { ... }
Matrix GetInverse() const { ... }

void   Transpose() { ... }
Matrix GetTranspose() const { ... }
```

Printing to console and working with files

```
//////////////////////                  I/O                  //////////////////////
void   Print(std::ostream& stream, int width, int precision) const { ... }
void   Print(std::ostream& stream, int width, int precision, Real zeroThreshold) const { ... }

friend std::ostream& operator<<(std::ostream& stream, const Matrix& a) { ... }
std::string to_string(int width, int precision) const { ... }

static bool LoadFromFile(std::string inFileName, Matrix& outMat) { ... }
static bool SaveToFile(const Matrix& mat, std::string inFileName) { ... }
```

MatrixNM<Type, N, M> - compile-time size

Different from Matrix in additional template parameters, and internal structure, but basically the same in functionality.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/MatrixNM.h

```
template <class Type, int N, int M>
class MatrixNM
{
public:
  Type _vals[N][M] = { {0} };

public:
  typedef Type value_type;      // make T available externally

  //////////////////////////            Constructors          //////////////////////////
  MatrixNM() {}
  MatrixNM(std::initializer_list<Type> values) { ... }
  MatrixNM(const MatrixNM& m) { ... }
  MatrixNM(const Type& m) { ... }
```

Extending BaseUtils with matrix helpers

Similarly to Vector helpers, we add to our BaseUtils.h header various helper for Matrix classes.

```
//////////////////            Creating Matrix from Vector             //////////////////
template<class Type> static Matrix<Type> RowMatrixFromVector(const Vector<Type>& b) { ... }
template<class Type> static Matrix<Type> ColumnMatrixFromVector(const Vector<Type>& b) { ... }
template<class Type> static Matrix<Type> DiagonalMatrixFromVector(const Vector<Type>& b) { ... }

template<class Type, int N> MatrixNM<Type, 1, N> RowMatrixFromVector(const VectorN<Type, N>& b) { ..
template<class Type, int N> MatrixNM<Type, N, 1> ColumnMatrixFromVector(const VectorN<Type, N>& b) {
template<class Type, int N> MatrixNM<Type, N, N> DiagonalMatrixFromVector(const VectorN<Type, N>& b)
```

```
//////////////////                  Matrix helpers                    //////////////////
template<class Type> static Matrix<Type> Commutator(const Matrix<Type>& a, const Matrix<Type>& b) { .
template<class Type> static Matrix<Type> AntiCommutator(const Matrix<Type>& a, const Matrix<Type>& b)

template<class Type>
static void MatrixDecomposeToSymAntisym(const Matrix<Type>& orig,
                                        Matrix<Type>& outSym,
                                        Matrix<Type>& outAntiSym) { ... }
```

```
//////////////////                Matrix functions                    //////////////////
template<class Type> static Matrix<Type> Exp(const Matrix<Type>& a, int n = 10) { ... }
template<class Type> static Matrix<Type> Sin(const Matrix<Type>& a, int n = 10) { ... }
template<class Type> static Matrix<Type> Cos(const Matrix<Type>& a, int n = 10) { ... }
```

Calculating different matrix properties

```
//////////////////                Real matrix helpers                  //////////////////
static bool IsOrthogonal(const Matrix<Real>& mat, double eps = Defaults::IsMatrixOrthogonalPrecision)

//////////////////               Complex matrix helpers                //////////////////
static Matrix<Real> GetRealPart(const Matrix<Complex>& a) { ... }
static Matrix<Real> GetImagPart(const Matrix<Complex>& a) { ... }

static Matrix<Complex> GetConjugateTranspose(const Matrix<Complex>& mat) { ... }
static Matrix<Complex> CmplxMatFromRealMat(const Matrix<Real>& mat) { ... }

static bool IsComplexMatReal(const Matrix<Complex>& mat) { ... }
static bool IsHermitian(const Matrix<Complex>& mat) { ... }
static bool IsUnitary(const Matrix<Complex>& mat) { ... }
```

Set of functions for performing arithmetic between Real and Complex matrices

```
//////////////////       Matrix<Complex> - Matrix<Real>  operations     //////////////////
static Matrix<Complex> AddMat(const Matrix<Complex>& a, const Matrix<Real>& b) { ... }
static Matrix<Complex> AddMat(const Matrix<Real>& a, const Matrix<Complex>& b) { ... }

static Matrix<Complex> SubMat(const Matrix<Complex>& a, const Matrix<Real>& b) { ... }
static Matrix<Complex> SubMat(const Matrix<Real>& a, const Matrix<Complex>& b) { ... }

static Matrix<Complex> MulMat(const Complex& a, const Matrix<Real>& b) { ... }
static Matrix<Complex> MulMat(const Matrix<Real>& a, const Complex& b) { ... }

static Matrix<Complex> MulMat(const Matrix<Complex>& a, const Matrix<Real>& b) { ... }
static Matrix<Complex> MulMat(const Matrix<Real>& a, const Matrix<Complex>& b) { ... }

static Vector<Complex> MulMatVec(const Matrix<Real>& a, const Vector<Complex>& b) { ... }
static Vector<Complex> MulMatVec(const Matrix<Complex>& a, const Vector<Real>& b) { ... }

static Vector<Complex> MulVecMat(const Vector<Complex>& a, const Matrix<Real>& b) { ... }
static Vector<Complex> MulVecMat(const Vector<Real>& a, const Matrix<Complex>& b) { ... }
```

Example usage

TODO – example with matrices

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_01_basic_objects/demo_matrix.cpp

FUNCTIONS AS FULL-FLEDGED OBJECTS

Function pointers have been in C and C++ from the start.

C++ introduced other options:

- Functors, ie. objects that overload operator()
- Using templates (with use of operator() overloading)
- lambdas

The most general approach is the one taken in Numerical Recipes in C++, where each "function" parameter is actually a template parameter, expecting only implementation of operator().

In an "opinionated" manner of our implementation, we will define a set of interfaces and treat functions as full-fledged objects

All interfaces are specified in IFunction.h header

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/interfaces/IFunction.h

General function definition:

```
template<typename _RetType, typename _ArgType>
class IFunction
{
public:
  virtual _RetType operator()(_ArgType) const = 0;

  virtual ~IFunction() {}
};
```

Breaking YAGNI because it is quite hard to envision any kind of function operating only on this interface, but … you never know.

RealFunction

Interface for real function is simple – real number in, real number out.

```
class IRealFunction : public IFunction<Real, Real>
{
public:
  virtual Real operator()(Real) const = 0;

  virtual ~IRealFunction() {}
};
```

Two implementations:

First one expects function pointer, meaning it will also accept lambda expression with function definition.

Make case for also defining std::function version … actually they will be essential for creating coordinate transformation classes that are not completely static (like spherical), but depend on some kind of parameter (angle for rotations, or matrix for general linear transformation)

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/Function.h

```
//////////////////////////     REAL FUNCTION      ////////////////////////////////////
class RealFunction : public IRealFunction
{
  Real(*_func)(const Real);
public:
  RealFunction(Real(*inFunc)(const Real)) : _func(inFunc) {}

  Real operator()(const Real x) const { return _func(x); }
};
class RealFunctionFromStdFunc : public IRealFunction
{
  std::function<Real(const Real)> _func;
public:
  RealFunctionFromStdFunc(std::function<Real(const Real)> inFunc) : _func(inFunc) {}

  Real operator()(const Real x) const { return _func(x); }
};
```

Example usage – real functions

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_01_basic_objects/demo_real_function.cpp

How to create them!

- Simple func pointer
- Class you cant change
    - std::func version
    - wrapper
- clas you can change – inherit IRealFunction

ScalarFunction<N>

Representing scalar function, that takes vector as an input, and returns real number.

Interface

```
//////////////////////////////////////////////////////////////////////
template<int N>
class IScalarFunction : public IFunction<Real, const VectorN<Real, N>&>
{
public:
  virtual Real operator()(const VectorN<Real, N>& x) const = 0;

  virtual ~IScalarFunction() {}
};
```

Implementations

```
//////////////////////////     SCALAR FUNCTION       ////////////////////////////////
template<int N>
class ScalarFunction : public IScalarFunction<N>
{
  Real(*_func)(const VectorN<Real, N>&);
public:
  ScalarFunction(Real(*inFunc)(const VectorN<Real, N>&)) : _func(inFunc) {}

  Real operator()(const VectorN<Real, N>& x) const { return _func(x); }
};

template<int N>
class ScalarFunctionFromStdFunc : public IScalarFunction<N>
{
  std::function<Real(const VectorN<Real, N>&)> _func;
public:
  ScalarFunctionFromStdFunc(std::function<Real(const VectorN<Real, N>&)> inFunc) : _func(inFunc) {}

  Real operator()(const VectorN<Real, N>& x) const { return _func(x); }
};
```

VectorFunction<N>

Representing vector function, that returns vector for given vector input parameter.

Interface

```
//////////////////////////////////////////////////////////////////////
template<int N>
class IVectorFunction : public IFunction<VectorN<Real, N>, const VectorN<Real, N>&>
{
public:
  virtual VectorN<Real, N> operator()(const VectorN<Real, N>& x) const = 0;

  virtual ~IVectorFunction() {}
};
```

Implementations

```
//////////////////////////    VECTOR FUNCTION N -> N      ////////////////////////////////
template<int N>
class VectorFunction : public IVectorFunction<N>
{
  VectorN<Real, N>(*_func)(const VectorN<Real, N>&);
public:
  VectorFunction(VectorN<Real, N>(*inFunc)(const VectorN<Real, N>&)) : _func(inFunc) {}

  VectorN<Real, N>     operator()(const VectorN<Real, N>& x) const { return _func(x); }
};

template<int N>
class VectorFunctionFromStdFunc : public IVectorFunction<N>
{
  std::function<VectorN<Real, N>(const VectorN<Real, N>&)> _func;
public:
  VectorFunctionFromStdFunc(std::function<VectorN<Real, N>(const VectorN<Real, N>&)>& inFunc)
    : _func(inFunc) {}

  VectorN<Real, N>     operator()(const VectorN<Real, N>& x) const { return _func(x); }
};
```

ParametricCurve<N>

Interfaces

Is it really needed to create this additional interface??? Hm …

```
//////////////////////////////////////////////////////////////////////
template<int N>
class IRealToVectorFunction : public IFunction<VectorN<Real, N>, Real>
{
public:
  virtual VectorN<Real, N> operator()(Real x) const = 0;

  virtual ~IRealToVectorFunction() {}
};
```

General interface for parametric curve.

```
//////////////////////////////////////////////////////////////////////
template<int N>
class IParametricCurve : public IRealToVectorFunction<N>
{
public:
  virtual Real getMinT() const = 0;
  virtual Real getMaxT() const = 0;

  std::vector<VectorN<Real, N>> GetTrace(double t1, double t2, int numPoints) const
  {
    std::vector<VectorN<Real, N>> ret;
    double deltaT = (t2 - t1) / (numPoints - 1);
    for (Real t = t1; t <= t2; t += deltaT)
      ret.push_back((*this)(t));
    return ret;
  }

  virtual ~IParametricCurve() {}
};
```

Implementation

```
template<int N>
class   ParametricCurve : public IParametricCurve<N>
{
  Real _minT;
  Real _maxT;
  VectorN<Real, N>(*_func)(Real);
public:
  ParametricCurve(VectorN<Real, N>(*inFunc)(Real))
    : _func(inFunc), _minT(Constants::NegativeInf), _maxT(Constants::PositiveInf) {}
  ParametricCurve(Real minT, Real maxT, VectorN<Real, N>(*inFunc)(Real))
    : _func(inFunc), _minT(minT), _maxT(maxT) {}

  Real getMinT() const { return _minT; }
  Real getMaxT() const { return _maxT; }

  virtual VectorN<Real, N> operator()(Real x) const { return _func(x); }
};
```

ParametricSurface<N>

Two different interfaces, for different kinds of surface models.

General surface

```
// complex surface, with fixed u limits, but variable w limits (dependent on u)
template<int N>
class IParametricSurface : public IFunction<VectorN<Real, N>, const VectorN<Real, 2>&>
{
public:
  virtual VectorN<Real, N> operator()(Real u, Real w) const = 0;

  virtual Real getMinU() const = 0;
  virtual Real getMaxU() const = 0;
  virtual Real getMinW(Real u) const = 0;
  virtual Real getMaxW(Real u) const = 0;

  virtual VectorN<Real, N> operator()(const VectorN<Real, 2>& coord) const
  {
    return operator()(coord[0], coord[1]);
  }

  virtual ~IParametricSurface() {}
};
```

Rectangular surface

```
// simple regular surface, defined on rectangular coordinate patch
template<int N>
class IParametricSurfaceRect : public IFunction<VectorN<Real, N>, const VectorN<Real, 2>&>
{
public:
  virtual VectorN<Real, N> operator()(Real u, Real w) const = 0;

  virtual Real getMinU() const = 0;
  virtual Real getMaxU() const = 0;
  virtual Real getMinW() const = 0;
  virtual Real getMaxW() const = 0;

  virtual VectorN<Real, N> operator()(const VectorN<Real, 2>& coord) const
  {
    return operator()(coord[0], coord[1]);
  }

  virtual ~IParametricSurfaceRect() {}
};
```

Example usage

Show examples of creating different kinds of:

- real function objects
- scalar function objects
- vector functions objects
- parametric curves
- parametric surfaces

Link to repo:

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_01_basic_objects/demo_functions.cpp

Functions test bed

BASIC GEOMETRY IN 2D AND 3D

Defining objects that will represent concepts from 2D and 3D analytical geometry – points, vectors, lines, planes.

Points

Why special classes for points, when they are structurally same as vectors, and position is mostly defined as “radius vector”?

Example, Cartesian point in 3D

Mostly trivial and expected operations for a point:

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/Geometry.h

```
class Point3Cartesian
{
private:
  Real _x, _y, _z;

public:
  Real  X() const { return _x; }
  Real& X()       { return _x; }
  Real  Y() const { return _y; }
  Real& Y()       { return _y; }
  Real  Z() const { return _z; }
  Real& Z()       { return _z; }

  Point3Cartesian() : _x(0), _y(0), _z(0) {}
  Point3Cartesian(Real x, Real y, Real z) : _x(x), _y(y), _z(z) {}

  Real  Dist(const Point3Cartesian& b) const { ... }

  bool  operator==(const Point3Cartesian& b) const { ... }
  bool  operator!=(const Point3Cartesian& b) const { ... }

  Point3Cartesian operator+(const Point3Cartesian& b) const { ... }
  Point3Cartesian operator*(Real b) { ... }
  Point3Cartesian operator/(Real b) { ... }

  friend Point3Cartesian operator*(Real a, const Point3Cartesian& b) { ... }
};
```

There’s a question of semantic validity of operator+() in this class, where adding two points is, even though mathematically precisely defined, something that looks off - namely, if you want to “add” to point, that is something you would do with a vector.

However, “adding” points is essential when we want to find middle points of a segment line or centroid of a triangle, and also for doing any kind of interpolation between two points, so it is part of the interface.

Vectors

Vector is NOT a point!

Point denotes position in space, in given coordinate system, vectors are … well, much more complex beasts

Cartesian vectors are simplest, as they are examples of *free vectors*, which are the same at every position in space, meaning that their component representation doesn't depend on position, unlike, for example vectors in spherical coordinate system.

We will deal with position-dependent kinds of vectors in Chapter 8 – Coordinate transformations, and for now cartesian vectors will do.

Example of Vector3Cartesian:

Specialized from VectorN, with Real type and dimension 3.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/VectorTypes.h

```
class Vector3Cartesian : public VectorN<Real, 3>
{
public:
  Real  X() const { return _val[0]; }
  Real& X()       { return _val[0]; }
  Real  Y() const { return _val[1]; }
  Real& Y()       { return _val[1]; }
  Real  Z() const { return _val[2]; }
  Real& Z()       { return _val[2]; }

  Vector3Cartesian() : VectorN<Real, 3>{ 0.0, 0.0, 0.0 } {}
  Vector3Cartesian(const VectorN<Real, 3>& b) : VectorN<Real, 3>{ b } {}
  Vector3Cartesian(Real x, Real y, Real z) : VectorN<Real, 3>{ x, y, z } {}
  Vector3Cartesian(std::initializer_list<Real> list) : VectorN<Real, 3>(list) { }
  Vector3Cartesian(const Point3Cartesian& a, const Point3Cartesian& b) { ... }

  Point3Cartesian getAsPoint() { ... }
```

Set of arithmetic operations

```
Point3Cartesian getAsPoint() { ... }

// For Cartesian vector, we will enable operator* to represent standard scalar product
Real operator*(const Vector3Cartesian& b) const { ... }

friend Vector3Cartesian operator*(const Vector3Cartesian& a, Real b) { ... }
friend Vector3Cartesian operator*(Real a, const Vector3Cartesian& b) { ... }
friend Vector3Cartesian operator/(const Vector3Cartesian& a, Real b) { ... }

friend Vector3Cartesian operator-(const Point3Cartesian& a, const Point3Cartesian& b)

friend Point3Cartesian operator+(const Point3Cartesian& a, const Vector3Cartesian& b)
friend Point3Cartesian operator-(const Point3Cartesian& a, const Vector3Cartesian& b)
```

And some more

```
bool IsParallelTo(const Vector3Cartesian& b, Real eps = 1e-15) const { ... }
bool IsPerpendicularTo(const Vector3Cartesian& b, Real eps = 1e-15) const { ... }
Real AngleToVector(const Vector3Cartesian& b) { ... }

friend Real ScalarProduct(const Vector3Cartesian& a, const Vector3Cartesian& b) { ... }
friend Vector3Cartesian VectorProd(const Vector3Cartesian& a, const Vector3Cartesian& b)
```

Lines

There are multiple ways of defining lines (TODO)

In both2D and 3D cases are specified with point, and vector representing direction, ie. representing parametric line equation.

Example Line3D

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/Geometry3D.h

```
class Line3D
{
private:
  Point3Cartesian  _point;
  Vector3Cartesian _direction;

public:
  Line3D() {}
  // by default, direction vector is normalized to unit vector (but, it need not be such!)
  Line3D(const Point3Cartesian& pnt, const Vector3Cartesian dir) { ... }
  Line3D(const Point3Cartesian& a, const Point3Cartesian& b) { ... }

  Point3Cartesian   StartPoint() const { return _point; }
  Point3Cartesian&  StartPoint() { return _point; }

  Vector3Cartesian  Direction() const { return _direction; }
  Vector3Cartesian& Direction() { return _direction; }

  Point3Cartesian operator()(Real t) const { return _point + t * _direction; }

  bool IsPerpendicular(const Line3D& b) const { ... }
  bool IsParallel(const Line3D& b) const { ... }
```

More operations:

```
// distance between point and line
Real Dist(const Point3Cartesian& pnt) const { ... }
// distance between two lines
Real Dist(const Line3D& line) const { ... }
// distance between two lines, while also returning nearest points on both lines
Real Dist(const Line3D& line, Point3Cartesian& out_line1_pnt, Point3Cartesian& out_line2_pnt) const

// nearest point on line to given point
Point3Cartesian NearestPointOnLine(const Point3Cartesian& pnt) const { ... }

// intersection of two lines
bool Intersection(const Line3D& line, Point3Cartesian& out_inter_pnt) const { ... }

// perpendicular line that goes through givenpoint
Line3D PerpendicularLineThroughPoint(const Point3Cartesian& pnt) { ... }
```

Polygons

BIG TODO!!!!

```
class Polygon2D
{
private:
  std::vector<Point2Cartesian> _points;
public:
  Polygon2D() {}
  Polygon2D(std::vector<Point2Cartesian> points) : _points(points) {}
  Polygon2D(std::initializer_list<Point2Cartesian> list) { ... }

  std::vector<Point2Cartesian>  Points() const { return _points; }
  std::vector<Point2Cartesian>& Points() { return _points; }

  Real Area() const { ... }

  bool IsSimple() const { ... }

  bool IsConvex() const { ... }

  std::vector<Triangle2D> Triangularization() const { ... }

  bool IsInside(Point2Cartesian pnt) const { ... }
};
```

Plane3D class

Representing plane in 3D Cartesian space, with general equation Ax + By + CZ + D = 0.

Extensive set of constructors,

```
class Plane3D
{
private:
  Real _A, _B, _C, _D;

public:
  Plane3D(const Point3Cartesian& a, const Vector3Cartesian& normal) { ... }
  Plane3D(const Point3Cartesian& a, const Point3Cartesian& b, const Point3Cartesian& c) { ... }
  // Hesse normal form
  Plane3D(Real alpha, Real beta, Real gamma, Real d) { ... }
  // segments on coordinate axes
  Plane3D(Real seg_x, Real seg_y, Real seg_z) { ... }

  static Plane3D GetXYPlane() { return Plane3D(Point3Cartesian(0, 0, 0), Vector3Cartesian(0, 0, 1)); }
  static Plane3D GetXZPlane() { return Plane3D(Point3Cartesian(0, 0, 0), Vector3Cartesian(0, 1, 0)); }
  static Plane3D GetYZPlane() { return Plane3D(Point3Cartesian(0, 0, 0), Vector3Cartesian(1, 0, 0)); }

  Real  A() const { return _A; }
  Real& A()       { return _A; }
  Real  B() const { return _B; }
  Real& B()       { return _B; }
  Real  C() const { return _C; }
  Real& C()       { return _C; }
  Real  D() const { return _D; }
  Real& D()       { return _D; }

  Vector3Cartesian  Normal() const { return Vector3Cartesian(_A, _B, _C); }
  Point3Cartesian   GetPointOnPlane() const { ... }

  void GetCoordAxisSegments(Real& outseg_x, Real& outseg_y, Real& outseg_z) { ... }
```

Basic operations between plane and points, lines, and other planes.

```
// point to plane operations
bool IsPointOnPlane(const Point3Cartesian& pnt, Real defEps = 1e-15) const { ... }
Real DistToPoint(const Point3Cartesian& pnt) const { ... }

Point3Cartesian ProjectionToPlane(const Point3Cartesian& pnt) const { ... }

// line to plane operations
bool IsLineOnPlane(const Line3D& line) const { ... }
Real AngleToLine(const Line3D& line) const { ... }
bool IntersectionWithLine(const Line3D& line, Point3Cartesian& out_inter_pnt) const { ..

// plane to plane operations
bool IsParallelToPlane(const Plane3D& plane) const { ... }
bool IsPerpendicularToPlane(const Plane3D& plane) const { ... }
Real AngleToPlane(const Plane3D& plane) const { ... }
Real DistToPlane(const Plane3D& plane) const { ... }
bool IntersectionWithPlane(const Plane3D& plane, Line3D& out_inter_line) const { ... }
```

Triangles

Each triangle has A(), B(), C() … but not enforced through interface

Basic (geometrical) triangle.

```
class Triangle
{
private:
  Real _a, _b, _c;

public:
  Real  A() const { return _a; }
  Real& A()       { return _a; }
  Real  B() const { return _b; }
  Real& B()       { return _b; }
  Real  C() const { return _c; }
  Real& C()       { return _c; }

  Triangle() : _a(0.0), _b(0.0), _c(0.0){}
  Triangle(Real a, Real b, Real c) : _a(a), _b(b), _c(c) {}

  Real Area() const { ... }

  bool IsRight() const { ... }
  bool IsIsosceles() const { ... }
  bool IsEquilateral() const { ... }
};
```

Triangle in 2D plane

```
class Triangle2D
{
private:
  Point2Cartesian _pnt1, _pnt2, _pnt3;

public:
  Triangle2D(Point2Cartesian pnt1, Point2Cartesian pnt2, Point2Cartesian pnt3)

  Real A() const { return _pnt1.Dist(_pnt2); }
  Real B() const { return _pnt2.Dist(_pnt3); }
  Real C() const { return _pnt3.Dist(_pnt1); }

  Point2Cartesian  Pnt1() const { return _pnt1; }
  Point2Cartesian& Pnt1()       { return _pnt1; }
  Point2Cartesian  Pnt2() const { return _pnt2; }
  Point2Cartesian& Pnt2()       { return _pnt2; }
  Point2Cartesian  Pnt3() const { return _pnt3; }
  Point2Cartesian& Pnt3()       { return _pnt3; }

  Real Area() const { ... }
};
```

Triangle in 3D space

```
class Triangle3D
{
private:
  Point3Cartesian _pnt1, _pnt2, _pnt3;
public:
  Triangle3D(Point3Cartesian pnt1, Point3Cartesian pnt2, Point3Cartesian pnt3)

  Real A() const { return _pnt1.Dist(_pnt2); }
  Real B() const { return _pnt2.Dist(_pnt3); }
  Real C() const { return _pnt3.Dist(_pnt1); }

  Point3Cartesian   Pnt1() const  { return _pnt1; }
  Point3Cartesian&  Pnt1()        { return _pnt1; }
  Point3Cartesian   Pnt2() const  { return _pnt2; }
  Point3Cartesian&  Pnt2()        { return _pnt2; }
  Point3Cartesian   Pnt3() const  { return _pnt3; }
  Point3Cartesian&  Pnt3()        { return _pnt3; }
};
```

Making Triangle3D a ParametricSurface<3>

So it can be used in surface integration routines!

Boxes

TODO!

Modeling surfaces & solid bodies

Two approaches

- Boundaries defined by functions
- Composed of surface polygons that (hopefully) enclose the body

Example usage - geometry

Analytical geometry examples in 2d and 3d

Creating bodies

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_01_basic_objects/demo_geometry.cpp

2. Basic algorithms

Putting some algorithmic meat on our objects.

SOLVING SYSTEMS OF LINEAR EQUATIONS

General introduction

Little bit of algebra – singular matrices, rank, kernel, null space

Gauss-Jordan

Starting with the basic Gauss-Jordan elimination algorithm … THAT SHOULD NEVER BE USED!

Gauss-Jordan with pivoting

This is what you should (at minimum) be using.

```
//////////////////////   GAUSS-JORDAN SOLVER    ////////////////////////////
template<class Type>
class GaussJordanSolver
{
public:
  // solving with Matrix RHS (ie. solving simultaneously for multiple RHS)
  static bool Solve(Matrix<Type>& a, Matrix<Type>& b) { ... }

  // solving for a given RHS vector
  static bool Solve(Matrix<Type>& a, Vector<Type>& b) { ... }

  // solving for a given RHS vector, but with return value
  // (in case of singular matrix 'a', exception is thrown)
  static Vector<Type> Solve(Matrix<Type>& a, const Vector<Type>& b) { ... }
};
```

LU decomposition

Solving iteratively – Jacoby, Gauss-Seidel, SOR

Example usage – solving systems of linear equations

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_lin_alg_sys_solvers.cpp

NUMERICAL DERIVATION

Most of the code here is based on Boost.Math toolkit, and its differentiation routines.

Mathematics of derivation approximation

Taylor expansion and all that jazz.

Derivation routines

Default step-sizes, defined in terms of machine Real type epsilon

```
static inline const Real NDer1_h = 2 * std::sqrt(Constants::Epsilon);
static inline const Real NDer2_h = std::pow(3 * Constants::Epsilon, 1.0 / 3.0);
static inline const Real NDer4_h = std::pow(11.25 * Constants::Epsilon, 1.0 / 5.0);
static inline const Real NDer6_h = std::pow(Constants::Epsilon / 168.0, 1.0 / 7.0);
static inline const Real NDer8_h = std::pow(551.25 * Constants::Epsilon, 1.0 / 9.0);
```

First order derivation of a RealFunction

```
static Real NDer1(const IRealFunction& f, Real x, Real h, Real* error = nullptr)
{
  Real yh = f(x + h);
  Real y0 = f(x);
  Real diff = yh - y0;
  if (error)
  {
    Real ym = f(x - h);
    Real ypph = std::abs(yh - 2 * y0 + ym) / h;

    // h*|f''(x)|*0.5 + (|f(x+h)+|f(x)|) * eps/h
    *error = ypph / 2 + (std::abs(yh) + std::abs(y0)) * Constants::Epsilon / h;
  }
  return diff / h;
}
```

We have quite a few versions:

```
static Real NDer1(const IRealFunction& f, Real x, Real h, Real* error = nullptr) { ... }
static Real NDer1(const IRealFunction& f, Real x, Real* error) { ... }
static Real NDer1(const IRealFunction& f, Real x) { ... }

static Real NDer1Left(const IRealFunction& f, Real x, Real* error = nullptr) { ... }
static Real NDer1Right(const IRealFunction& f, Real x, Real* error = nullptr) { ... }
static Real NDer1Left(const IRealFunction& f, Real x, Real h, Real* error = nullptr) { ... }
static Real NDer1Right(const IRealFunction& f, Real x, Real h, Real* error = nullptr) { ... }

static Real NSecDer1(const IRealFunction& f, Real x, Real h, Real* error = nullptr) { ... }
static Real NSecDer1(const IRealFunction& f, Real x, Real* error = nullptr) { ... }

static Real NThirdDer1(const IRealFunction& f, Real x, Real h, Real* error = nullptr) { ... }
static Real NThirdDer1(const IRealFunction& f, Real x, Real* error = nullptr) { ... }
```

Calculating fourth order derivation

```
static Real NDer4(const IRealFunction& f, Real x, Real h, Real* error = nullptr)
{
  Real yh = f(x + h);
  Real ymh = f(x - h);
  Real y2h = f(x + 2 * h);
  Real ym2h = f(x - 2 * h);

  Real y2 = ym2h - y2h;
  Real y1 = yh - ymh;

  if (error)
  {
    // ...
    Real y3h = f(x + 3 * h);
    Real ym3h = f(x - 3 * h);

    // Error from fifth derivative:
    *error = std::abs((y3h - ym3h) / 2 + 2 * (ym2h - y2h) + 5 * (yh - ymh) / 2) / (30 * h);
    // Error from function evaluation:
    *error += Constants::Epsilon * (std::abs(y2h) + std::abs(ym2h) + 8 * (std::abs(ymh) + std::abs(yh))) / (12 * h)
  }
  return (y2 + 8 * y1) / (12 * h);
}
```

Fourth order (partial) derivation of ScalarFunction (we have to specify by which index we are deriving)

```
template <int N>
static Real NDer4Partial(const IScalarFunction<N>& f, int deriv_index, const VectorN<Real, N>& point, Real h, Real* error = nullptr)
{
  Real     orig_x = point[deriv_index];

  VectorN<Real, N> x{ point };
  x[deriv_index] = orig_x + h;
  Real yh = f(x);

  x[deriv_index] = orig_x - h;
  Real ymh = f(x);

  x[deriv_index] = orig_x + 2 * h;
  Real y2h = f(x);

  x[deriv_index] = orig_x - 2 * h;
  Real ym2h = f(x);

  Real y2 = ym2h - y2h;
  Real y1 = yh - ymh;

  if (error)
  {
    x[deriv_index] = orig_x + 3 * h;
    Real y3h = f(x);

    x[deriv_index] = orig_x - 3 * h;
    Real ym3h = f(x);

    *error = std::abs((y3h - ym3h) / 2 + 2 * (ym2h - y2h) + 5 * (yh - ymh) / 2) / (30 * h);
    *error += Constants::Epsilon * (std::abs(y2h) + std::abs(ym2h) + 8 * (std::abs(ymh) + std::abs(yh))) / (12 * h);
  }
  return (y2 + 8 * y1) / (12 * h);
}
```

Vector function derivation routines, fourth order

```
template <int N>
static Real NDer4Partial(const IVectorFunction<N>& f, int func_index, int deriv_index, const VectorN<Real, N>& point,
                         Real* error = nullptr) { ... }
template <int N>
static Real NDer4Partial(const IVectorFunction<N>& f, int func_index, int deriv_index, const VectorN<Real, N>& point, Real h,
                         Real* error = nullptr) { ... }
template <int N>
static VectorN<Real, N> NDer4PartialByAll(const IVectorFunction<N>& f, int func_index, const VectorN<Real, N>& point,
                                          VectorN<Real, N>* error = nullptr) { ... }
template <int N>
static VectorN<Real, N> NDer4PartialByAll(const IVectorFunction<N>& f, int func_index, const VectorN<Real, N>& point, Real h,
                                          VectorN<Real, N>* error = nullptr) { ... }
template <int N>
static MatrixNM<Real, N, N> NDer4PartialAllByAll(const IVectorFunction<N>& f, const VectorN<Real, N>& point,
                                                 MatrixNM<Real, N, N>* error = nullptr) { ... }
template <int N>
static MatrixNM<Real, N, N> NDer4PartialAllByAll(const IVectorFunction<N>& f, const VectorN<Real, N>& point, Real h,
                                                 MatrixNM<Real, N, N>* error = nullptr) { ... }
```

ParametricCurve derivations

```
template <int N>
static VectorN<Real, N> NDer4(const IParametricCurve<N>& f, Real t, Real* error = nullptr) { ... }

template <int N>
static VectorN<Real, N> NDer4(const IParametricCurve<N>& f, Real t, Real h, Real* error = nullptr) { ..

template <int N>
static VectorN<Real, N> NSecDer4(const IParametricCurve<N>& f, Real t, Real* error = nullptr) { ... }

template <int N>
static VectorN<Real, N> NSecDer4(const IParametricCurve<N>& f, Real t, Real h, Real* error = nullptr) {


template <int N>
static VectorN<Real, N> NThirdDer4(const IParametricCurve<N>& f, Real t, Real* error = nullptr) { ... }

template <int N>
static VectorN<Real, N> NThirdDer4(const IParametricCurve<N>& f, Real t, Real h, Real* error = nullptr)
```

Examples of usage - derivation

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_derivation.cpp

Automatic differentiation

Short overview of basic AD implementation

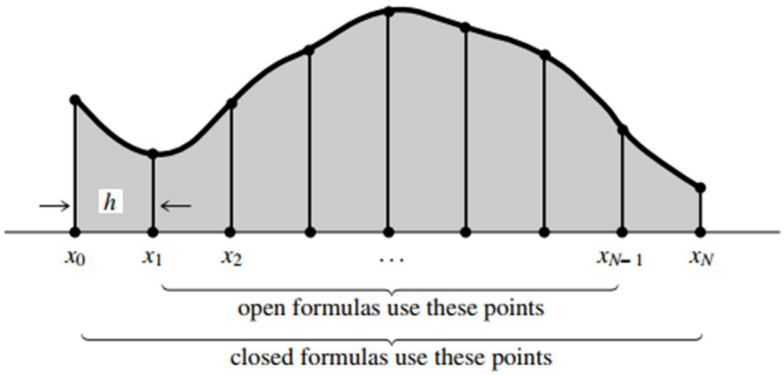
NUMERICAL INTEGRATION

We want so solve this

$$I = \int_a^b f(x)dx$$

For a given function f, and limits a and b.

Open vs closed formulas.



Basic trapezoidal rule for approximation in single interval.

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{1}{2}f_0 + \frac{1}{2}f_1\right] \quad + O(h^3 f'')$$

Simpson's rule

$$\int_{x_0}^{x_2} f(x)dx = h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{1}{3}f_2\right] \quad + O(h^5 f^{(4)})$$

Extended trapezoidal rule.

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-2} + \frac{1}{2}f_{N-1}\right] \quad + O\left(\frac{(b-a)^3 f''}{N^2}\right)$$

Trapezoidal integrator

Basic interface, because all integrators are “refinement machines”, calculating with more and more interior points, until required accuracy is achieved.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/core/Integration.h

```
enum IntegrationMethod { TRAP, SIMPSON, ROMBERG, GAUSS10 };

class IQuadrature {
public:
  int _currStep;
  virtual Real next() = 0;
};
```

Implementation of trapezoidal refinement step

```
class TrapIntegrator : IQuadrature
{
public:
  Real _a, _b, _currSum;
  const IRealFunction& _func;

  TrapIntegrator(const IRealFunction& func, Real aa, Real bb) :
    _func(func), _a(aa), _b(bb), _currSum(0.0)  { _currStep = 0; }

  // ...
  Real next() {
    Real x, sum, del;
    int subDivNum, j;

    _currStep++;
    if (_currStep == 1) {
      return (_currSum = 0.5 * (_b - _a) * (_func(_a) + _func(_b)));
    }
    else {
      for (subDivNum = 1, j = 1; j < _currStep - 1; j++)
        subDivNum *= 2;

      del = (_b - _a) / subDivNum;
      x = _a + 0.5 * del;

      for (sum = 0.0, j = 0; j < subDivNum; j++, x += del)
        sum += _func(x);

      _currSum = 0.5 * (_currSum + (_b - _a) * sum / subDivNum);

      return _currSum;
    }
  }
};
```

Implementation

```
static Real IntegrateTrap(const IRealFunction& func, Real a, Real b,
                          int* doneSteps, Real* achievedPrec,
                          const Real eps = Defaults::TrapezoidIntegrationEPS)
{
  Real currSum, oldSum = 0.0;

  TrapIntegrator t(func, a, b);

  for (int j = 0; j < Defaults::TrapezoidIntegrationMaxSteps; j++)
  {
    currSum = t.next();

    if (j > 5) {
      if ( std::abs(currSum - oldSum) < eps * std::abs(oldSum) ||
           (currSum == 0.0 && oldSum == 0.0) )
      {
        if (doneSteps != nullptr)     *doneSteps = j;
        if (achievedPrec != nullptr)  *achievedPrec = std::abs(currSum - oldSum);

        return currSum;
      }
    }

    oldSum = currSum;
  }
  if (doneSteps != nullptr)     *doneSteps = Defaults::TrapezoidIntegrationMaxSteps;
  if (achievedPrec != nullptr)  *achievedPrec = std::abs(currSum - oldSum);

  return currSum;
}
```

Gauss-Legendre integration

```
static Real IntegrateGauss10(const IRealFunction& func, const Real a, const Real b)
{
  // Returns the integral of the function func between a and b, by ten-point GaussLegendre integration:
  // the function is evaluated exactly ten times at interior points in the range of integration.
  static const Real x[] = { 0.1488743389816312,0.4333953941292472,
                            0.6794095682990244,0.8650633666889845,0.9739065285171717 };
  static const Real w[] = { 0.2955242247147529,0.2692667193099963,
                            0.2190863625159821,0.1494513491505806,0.0666713443086881 };

  Real xm = 0.5 * (b + a);
  Real xr = 0.5 * (b - a);

  Real s = 0;
  for (int j = 0; j < 5; j++)
  {
    Real dx = xr * x[j];
    s += w[j] * (func(xm + dx) + func(xm - dx));
  }
  return s *= xr;
}
```

Examples of usage – integrating real function

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_integration.cpp

Integrating in 2D

Helper function, for integrating inner loop

```
struct SurfaceIntegralInner : public IRealFunction
{
  mutable Real        _currX;
  IScalarFunction<2>& _funcToIntegrate;

  SurfaceIntegralInner(IScalarFunction<2>& func) : _funcToIntegrate(func) {}

  Real operator()(const Real y) const
  {
    VectorN<Real, 2> v{ _currX, y };
    return _funcToIntegrate(v);
  }
};
```

Helper function for outer loop

```
struct SurfaceIntegralOuter : public IRealFunction
{
  mutable SurfaceIntegralInner _fInner;

  IntegrationMethod     _integrMethod;
  IScalarFunction<2>&   _funcToIntegrate;

  Real(*_yRangeLow)(Real);
  Real(*_yRangeUpp)(Real);

  SurfaceIntegralOuter(IScalarFunction<2>& func, IntegrationMethod inMethod,
                       Real yy1(Real), Real yy2(Real))
     : _yRangeLow(yy1), _yRangeUpp(yy2),
       _fInner(func), _funcToIntegrate(func), _integrMethod(inMethod)  { }

  // for given x, will return (ie. integrate function over Y-range)
  Real operator()(const Real x) const
  {
    _fInner._currX = x;
    switch (_integrMethod)
    {
      case SIMPSON:
        return IntegrateSimpson(_fInner, _yRangeLow(x), _yRangeUpp(x));
        // ...
      case GAUSS10:
        return IntegrateGauss10(_fInner, _yRangeLow(x), _yRangeUpp(x));
      default:
        return IntegrateTrap(_fInner, _yRangeLow(x), _yRangeUpp(x));
    }
  }
};
```

And finally, main integrator

```
static Real IntegrateSurface(IScalarFunction<2>& func, IntegrationMethod method,
                             const Real x1, const Real x2,
                             Real y1(Real), Real y2(Real))
{
  SurfaceIntegralOuter f1(func, method, y1, y2);

  switch (method)
  {
    case SIMPSON:
      return IntegrateSimpson(f1, x1, x2);
    // ...
    case GAUSS10:
      return IntegrateGauss10(f1, x1, x2);
    default:
      return IntegrateTrap(f1, x1, x2);
  }
}
```

Examples of usage – integrating in 2D

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_integration_2d.cpp

Integrating in 3D

Helper function, for innermost loop integration

```
struct VolumeIntegralInnermost : public IRealFunction
{
  mutable Real          _currX, _currY;
  IScalarFunction<3>&   _funcToIntegrate;

  VolumeIntegralInnermost(IScalarFunction<3>& func)
    : _funcToIntegrate(func), _currX{ 0 }, _currY{ 0 } {}

  Real operator()(const Real z) const
  {
    VectorN<Real, 3> v{ _currX, _currY, z };

    return _funcToIntegrate(v);
  }
};
```

Inner integration

```
struct VolumeIntegralInner : public IRealFunction
{
  mutable VolumeIntegralInnermost _fInnermost;

  IScalarFunction<3>& _funcToIntegrate;

  Real(*_zRangeLow)(Real, Real);
  Real(*_zRangeUpp)(Real, Real);

  VolumeIntegralInner(IScalarFunction<3>& func,
                      Real zz1(Real, Real), Real zz2(Real, Real))
    : _zRangeLow(zz1), _zRangeUpp(zz2), _funcToIntegrate(func), _fInnermost(func) {}

  Real operator()(const Real y) const
  {
    _fInnermost._currY = y;

    return IntegrateGauss10(_fInnermost,
                            _zRangeLow(_fInnermost._currX, y),
                            _zRangeUpp(_fInnermost._currX, y));
  }
};
```

Outer integration loop

```
struct VolumeIntegralOuter : public IRealFunction
{
  mutable VolumeIntegralInner _fInner;

  IScalarFunction<3>& _funcToIntegrate;
  Real(*_yRangeLow)(Real);
  Real(*_yRangeUpp)(Real);

  VolumeIntegralOuter(IScalarFunction<3>& func, Real yy1(Real), Real yy2(Real),
                      Real z1(Real, Real), Real z2(Real, Real))
    : _yRangeLow(yy1), _yRangeUpp(yy2), _funcToIntegrate(func), _fInner(func, z1, z2)
  {
  }

  Real operator()(const Real x) const
  {
    _fInner._fInnermost._currX = x;

    return IntegrateGauss10(_fInner, _yRangeLow(x), _yRangeUpp(x));
  }
};
```

And finally, volume integrator

Sets up needed objects, and fires it all up.

```
static Real IntegrateVolume(IScalarFunction<3>& func,
                            const Real x1, const Real x2,
                            Real y1(Real), Real y2(Real),
                            Real z1(Real, Real), Real z2(Real, Real))
{
  VolumeIntegralOuter f1(func, y1, y2, z1, z2);

  return IntegrateGauss10(f1, x1, x2);
}
```

Examples of usage – integrating in 3D

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_integration_3d.cpp

INTERPOLATING FUNCTIONS

TODO – INTRO

Base class for interpolating real functions

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/InterpolatedFunction.h

```
class RealFunctionInterpolated : public IRealFunction
{
private:
  int   _numPoints, _usedPoints;
  Vector<Real> _x, _y;            // we are storing copies of given values!

public:
  RealFunctionInterpolated(const Vector<Real> &x, const Vector<Real> &y,
                           int usedPointsInInterpolation)
          :  _x(x), _y(y), _numPoints(x.size()), _usedPoints(usedPointsInInterpolation)
  {
    // throw if not enough points
    if (_numPoints < 2 || _usedPoints < 2 || _usedPoints > _numPoints)
      throw RealFuncInterpInitError("RealFunctionInterpolated size error");
  }

  virtual ~RealFunctionInterpolated() {}

  Real virtual calcInterpValue(int startInd, Real x) const = 0;

  inline Real MinX() const { return X(0); }
  inline Real MaxX() const { return X(_numPoints-1); }

  inline Real X(int i) const { return _x[i]; }
  inline Real Y(int i) const { return _y[i]; }

  inline int  getNumPoints() const { return _numPoints; }
  inline int  getInterpOrder() const { return _usedPoints; }

  Real operator()(Real x) const
  {
    int startInd = locate(x);
    return calcInterpValue(startInd, x);
  }

  // Given a value x, return a value j such that x is (insofar as possible) centered in the subrange
  // xx[j..j+mm-1], where xx is the stored pointer. The values in xx must be monotonic, either
  // increasing or decreasing. The returned value is not less than 0, nor greater than _numPoints-1.
  int locate(const Real x) const { ... }
};
```

LinearInterpRealFunc

```
class LinearInterpRealFunc : public RealFunctionInterpolated
{
public:
  LinearInterpRealFunc(const Vector<Real>& xv, Vector<Real>& yv)
        : RealFunctionInterpolated(xv, yv, 2) {}

  Real calcInterpValue(int j, Real x) const {
    if (X(j) == X(j + 1))
      return Y(j);
    else
      return Y(j) + ((x - X(j)) / (X(j + 1) - X(j)) * (Y(j + 1) - Y(j)));
  }
};
```

PolynomInterpFunc

```
// Polynomial interpolation object.Construct with x and y vectors, and the number M of points
// to be used locally(polynomial order plus one), then call interp for interpolated values.
class PolynomInterpRealFunc : public RealFunctionInterpolated
{
private:
  mutable Real _errorEst;
public:
  PolynomInterpRealFunc(const Vector<Real>& xv, Vector<Real>& yv, int m)
        : RealFunctionInterpolated(xv, yv, m), _errorEst(0.) { }

  Real getLastErrorEst() const { return _errorEst; }

  // Given a value x, and using pointers to data xx and yy, this routine returns an interpolated
  // value y, and stores an error estimate _errorEst.The returned value is obtained by mm-point polynomial
  // interpolation on the subrange xx[startInd..startInd + mm - 1].
  Real calcInterpValue(int startInd, Real x) const { ... }
};
```

SplineInterpFunc

```
// Cubic spline interpolation object. Construct with x and y vectors, and (optionally) values of
// the first derivative at the endpoints, then call interp for interpolated values.
struct SplineInterpRealFunc : RealFunctionInterpolated
{
  Vector<Real> _secDerY;

  SplineInterpRealFunc(Vector<Real>& xv, Vector<Real>& yv, Real yp1 = 1.e99, Real ypn = 1.e99)
    : RealFunctionInterpolated(xv, yv, 2), _secDerY(xv.size())
  {
    initSecDerivs(&xv[0], &yv[0], yp1, ypn);
  }

  // This routine stores an array _secDerY[0..numPoints-1] with second derivatives of the interpolating function
  // at the tabulated points pointed to by xv, using function values pointed to by yv. If yp1 and/or
  // ypn are equal to 1  1099 or larger, the routine is signaled to set the corresponding boundary
  // condition for a natural spline, with zero second derivative on that boundary; otherwise, they are
  // the values of the first derivatives at the endpoints.
  void initSecDerivs(const Real* xv, const Real* yv, Real yp1, Real ypn) { ... }

  // Given a value x, and using pointers to data xx and yy, and the stored vector of second derivatives
  // _secDerY, this routine returns the cubic spline interpolated value y.
  Real calcInterpValue(int startInd, Real x) const { ... }
};
```

ParametricCurveSplineInterp

```
// Object for interpolating a curve specified by _numPoints points in N dimensions.
template<int N>
class SplineInterpParametricCurve : public IParametricCurve<N>
{
  Real _minT, _maxT;
  int  _dim, _numPoints, _bemba;
  bool _isCurveClosed;

  Matrix<Real> _curvePoints;
  Vector<Real> s;
  Vector<Real> ans;

  std::vector<SplineInterpRealFunc*> srp;

public:
  // Constructor. The _numPoints   _dim matrix ptsin inputs the data points. Input close as 0 for
  // an open curve, 1 for a closed curve. (For a closed curve, the last data point should not
  // duplicate the first - the algorithm will connect them.)
  SplineInterpParametricCurve(Real minT, Real maxT, const Matrix<Real>& ptsin, bool close = 0) { ... }

  SplineInterpParametricCurve(const Matrix<Real>& ptsin, bool close = 0) { ... }

  ~SplineInterpParametricCurve() { ... }

  Real getMinT() const { return _minT; }
  Real getMaxT() const { return _maxT; }

  // Interpolate a point on the stored curve. The point is parameterized by t, in the range [0,1].
  // For open curves, values of t outside this range will return extrapolations (dangerous!). For
  // closed curves, t is periodic with period 1
  VectorN<Real, N> operator()(Real t) const { ... }

  // ...
  Real fprime(Real* x, Real* y, int pm) { ... }

  Real rad(const Real* p1, const Real* p2) { ... }
};
```

Example usage – interpolating functions

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_interpolation.cpp

ROOT FINDING

Polynomic roots up to 4th order

```
// a * x^2 + b * x + c = 0
static int  SolveQuadratic(Real a, Real b, Real c,
                           Complex& x1, Complex& x2) { ... }
static void SolveQuadratic(const Complex &a, const Complex &b, const Complex &c,
                           Complex& x1, Complex& x2) { ... }
// CHECK!!!
static void SolveCubic(Real a, Real b, Real c, Real d,
                       Complex& x1, Complex& x2, Complex& x3) { ... }
// TODO!!!
static void SolveQuartic(Real a, Real b, Real c, Real d, Real e,
                         Complex& x1, Complex& x2, Complex& x3, Complex& x4) { ... }
```

Bracketing roots

```
// Given a function func and an initial guessed range x1 to x2, the routine expands
// the range geometrically until a root is bracketed by the returned values x1 and x2 (in which
// case function returns true) or until the range becomes unacceptably large (in which case we
// return false).
static bool BracketRoot(const IRealFunction& func, double& x1, double& x2, int MaxTry = 50) { ... }

// Given a function fx defined on the interval[x1, x2], subdivide the interval into
// _numPoints equally spaced segments, and search for zero crossings of the function.numRoots will be set
// to the number of bracketing pairs found.If it is positive, the vectors xb1[0..numRoots - 1] and
// xb2[0..numRoots - 1] will be filled sequentially with any bracketing pairs that are found.On input,
// these vectors may have any size, including zero; they will be resized to   numRoots.
static void FindRootBrackets(const IRealFunction& func, const Real x1, const Real x2, const int numPoints,
                             Vector<Real>& xb1, Vector<Real>& xb2, int& numRoots) { ... }
```

Bisection

```
// Using bisection, return the root of a function or functor func known to lie between x1 and x2.
// The root will be refined until its accuracy is xacc.
static Real FindRootBisection(const IRealFunction& func, Real x1, Real x2, Real xacc) { ... }
```

Newton-Raphson algorithm

```
// Using the Newton-Raphson method, return the root of a function known to lie in the interval
// x1; x2.The root will be refined until its accuracy is known within ˙xacc.
static Real FindRootNewton(const IRealFunction& func, Real x1, Real x2, Real xacc)
{
  Real rtn = 0.5 * (x1 + x2);
  for (int j = 0; j < Defaults::NewtonRaphsonMaxSteps; j++)
  {
    Real f  = func(rtn);
    Real df = Derivation::NDer4(func, rtn);

    Real dx = f / df;

    rtn -= dx;

    if ((x1 - rtn) * (rtn - x2) < 0.0)
      throw RootFindingError("Jumped out of brackets in FindRootNewton");

    if (std::abs(dx) < xacc)
      return rtn;
  }
  throw RootFindingError("Maximum number of iterations exceeded in FindRootNewton");
}
```

Example usage – root finding

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_root_finding.cpp

3. Visualization, anyone?

Using Qt

Ideal for open-source projects

Widgets, enabling interactive application

2D animation for collision simulator

Creating simple application for visualization of moving circles within given rectangle.

We will be extending this in Chapter 4, for collision simulator.

Using FLTK

Simple to use, and cross-platform, FLTK is another option, especially if Qt open source (or commercial) license is not suitable.

Visualizing real function

Short example of visualizing real function in some interval.

Using Python – Mathplotlib or VTK binding?

Matplotlib is beautiful piece of software, and by creating interface between our C++ code and Python, we can efficiently utilize it.

Based on book “Introduction to numerical programming” by Titus Adrian Beu and toolkit presented there.

Contour plot

USING GNUPLOT?

To do, or not to do?

It is cross-platform, and powerful in its visualization capabilities

USING .NET WPF

Unlike previous visualization solutions, this one is available only for Windows, as it relies on mighty Windows Presentation Foundation framework for visualization.

It is similar in its usage pattern to GnuPlot where we need to export all relevant data that needs visualizing in a file, and then give that file as an input to WPF visualizing application.

WPF visualizers are realized as a set of couple of distinct applications, with each one of them focusing on one type of visualization. So far, following visualizators are available:

1) RealFunctionVisualizer (with capability of visualizing multiple functions)
2) ScalarFunction2Visualizer – for visualizing 2D surfaces given as z=f(x,y) on rectangular patch
3) VectorFieldVisualizer – for visualizing vector field in 3D
4) ParametricCurve3Visualizer

Relying on Serializer class

Functionality for serializing real functions and parametric curves

```
class Serializer
{
public:
  // Real function serialization
  static bool SaveRealMultiFunc(std::vector<IRealFunction*> funcs, std::string title,
                                Real x1, Real x2, int count, std::string fileName) { ... }

  static bool SaveRealFuncEquallySpaced(const IRealFunction& f, std::string title,
                                        Real x1, Real x2, int numPoints, std::string fileName) { ... }

  static bool SaveRealFuncEquallySpacedDetailed(const IRealFunction& f, std::string title,
                                                Real x1, Real x2, int numPoints, std::string fileName) { ... }

  static bool SaveRealFuncVariableSpaced(const IRealFunction& f, std::string title,
                                         Vector<Real> points, std::string fileName) { ... }

  // Parametric curve serialization
  template<int N>
  static bool SaveParamCurve(const IRealToVectorFunction<N>& f, std::string inType, std::string title,
                             Real t1, Real t2, int numPoints, std::string fileName) { ... }

  template<int N>
  static bool SaveParamCurve(const IRealToVectorFunction<N>& f, std::string inType, std::string title,
                             Vector<Real> points, std::string fileName) { ... }

  template<int N>
  static bool SaveAsParamCurve(std::vector<VectorN<Real, N>> vals, std::string inType, std::string title,
                               Real t1, Real t2, int numPoints, std::string fileName) { ... }

  // ...
  static bool SaveParamCurveCartesian3D(const IRealToVectorFunction<3>& f, std::string title,
                                        Real t1, Real t2, int numPoints, std::string fileName) { ... }
```

Functionality for serializing scalar and vector functions

```
// Scalar function serialization
static bool SaveScalarFunc2DCartesian(const IScalarFunction<2>& f, std::string title,
                                      Real x1, Real x2, int numPointsX,
                                      Real y1, Real y2, int numPointsY, std::string fileName) { ... }

static bool SaveScalarFunc3DCartesian(const IScalarFunction<3>& f, std::string title,
                                      Real x1, Real x2, int numPointsX,
                                      Real y1, Real y2, int numPointsY,
                                      Real z1, Real z2, int numPointsZ, std::string fileName) { ... }

// vector function serialization
static bool SaveVectorFunc3D(const IVectorFunction<3>& f, std::string inType, std::string title,
                             Real x1_start, Real x1_end, int numPointsX1,
                             Real x2_start, Real x2_end, int numPointsX2,
                             Real x3_start, Real x3_end, int numPointsX3, std::string fileName) { ... }

static bool SaveVectorFunc3D(const IVectorFunction<3>& f, std::string inType, std::string title,
                             Real x1_start, Real x1_end, int numPointsX1,
                             Real x2_start, Real x2_end, int numPointsX2,
                             Real x3_start, Real x3_end, int numPointsX3,
                             std::string fileName, Real upper_threshold) { ... }

static bool SaveVectorFunc3DCartesian(const IVectorFunction<3>& f, std::string title,
                                      Real x1, Real x2, int numPointsX,
                                      Real y1, Real y2, int numPointsY,
                                      Real z1, Real z2, int numPointsZ, std::string fileName) { ... }
static bool SaveVectorFunc3DCartesian(const IVectorFunction<3>& f, std::string title,
                                      Real x1, Real x2, int numPointsX,
                                      Real y1, Real y2, int numPointsY,
                                      Real z1, Real z2, int numPointsZ, std::string fileName, Real upper_threshold)
```

Visualizing real functions

RealFunctionVisualizer is implemented as a .NET Core 8 WPF application.

Simple structure – it loads all given input files, does necessary analysis of ranges and values, and then visualizes all data on a Canvas.

Supported input types:

REAL_FUNCTION

Example format of text file:

```
REAL_FUNCTION
x1: 0
x2: 10
NumPoints: 500
0 0
0.0200401 0.0245782
0.0400802 0.0491564
0.0601202 0.0737346
0.0801603 0.0983128
…
9.8998 -23.1583
9.91984 -24.0757
9.93988 -24.993
9.95992 -25.9104
9.97996 -26.8277
10 -27.7451
```

REAL_FUNCTION_EQUALLY_SPACED

Similar, but has only one value in a row since ‘x’ value is calculated from x1, x2 and NumPoints parameters.

MULTI_REAL_FUNCTION

Parameters are: graph title, number of functions (and expected values in the row), number of points, x1, x2.

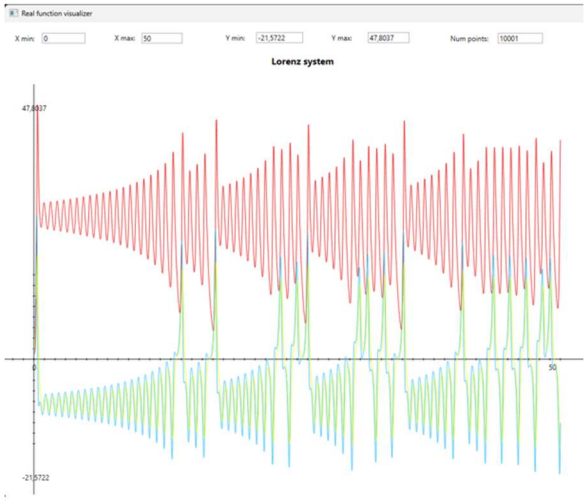
Example of a Lorenz system solution:

```
MULTI_REAL_FUNCTION_VARIABLE_SPACED
Lorenz system
3
10001
0
50
0 2 1 1
0.005 1.95768 1.26133 0.997865
0.01 1.93002 1.51666 0.998054
0.015 1.91584 1.76819 1.00049
0.02 1.91433 2.01768 1.0052
0.025 1.92502 2.26652 1.01229
0.03 1.94747 2.51609 1.0219
```

```
…
49.975 -16.5877 -17.0318 37.8714
49.98 -16.585 -16.113 38.7266
49.985 -16.5356 -15.1326 39.4912
49.99 -16.4396 -14.0995 40.1596
49.995 -16.2973 -13.0231 40.7272
50 -16.1097 -11.9137 41.1906
```

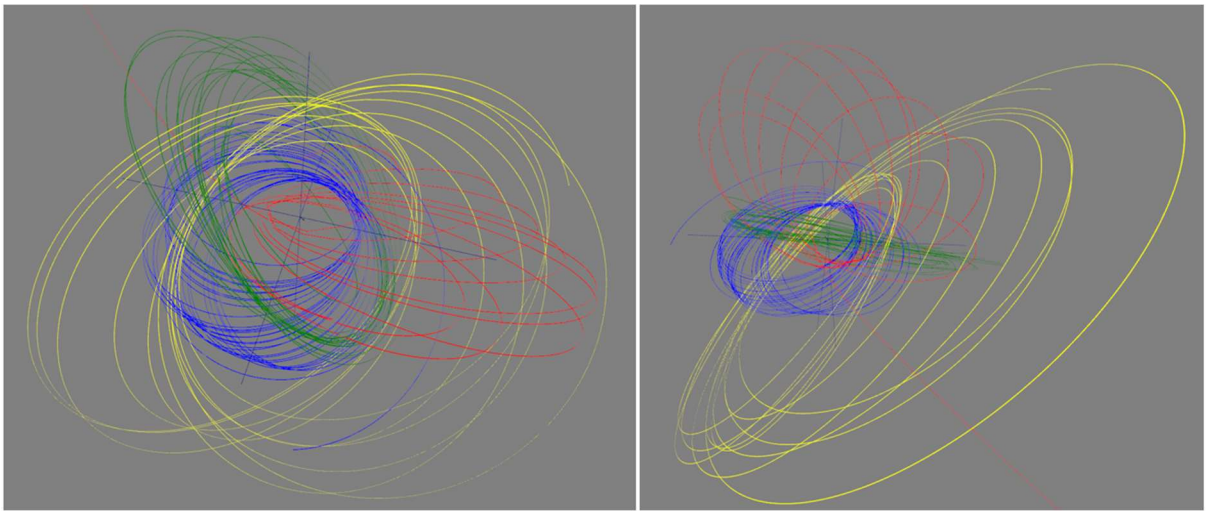
Can be given multiple inputs

Solution of Lorentz system.



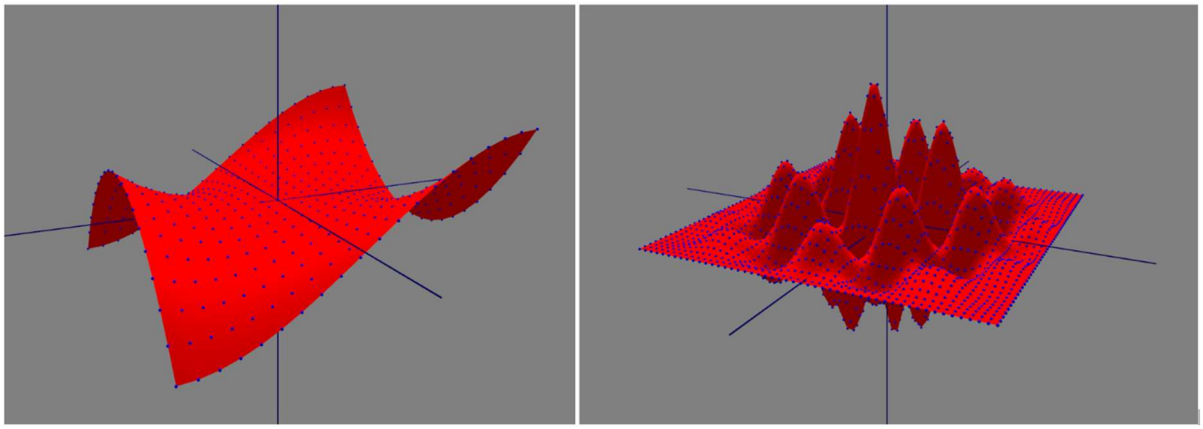
Visualizing parametric curves in 2D and 3D

Visualizing 3D trajectories for 5-body gravity problem (fifth body is a massive one in the center, so it barely moves during simulation).

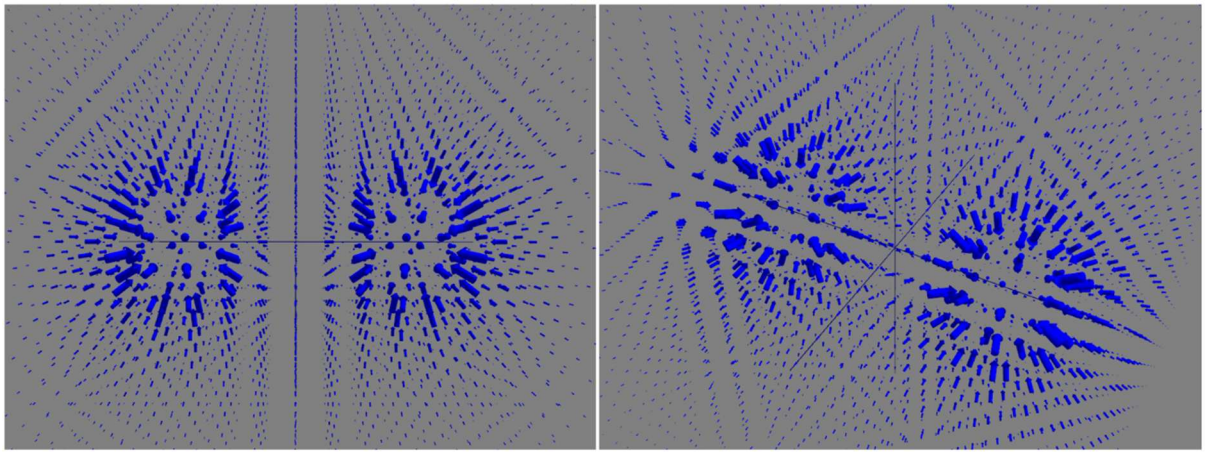


Visualizer for parametric curves also has animation capability, where each curve is visualized by a moving little sphere, following curve progress.

Visualizing surfaces



Visualizing 3D vector field



Visualization objects in C++ for WPF visualizer

Set of objects that removes from user need for manually saving data in a file, and then giving that file as parameter.

Basically, it uses existing Serializer object that can serialize to a file values for different kinds of functions, defines some constants with paths to visualizing applications, and ties all that in a set of simple static functions

Based on a set of global paths.

```
// Global paths for Visualizers
static const std::string MML_GLOBAL_PATH = "E:\\Projects\\MinimalMathLibrary";

static const std::string MML_PATH_ResultFiles = MML_GLOBAL_PATH +
               "\\results\\";
static const std::string MML_PATH_RealFuncViz = MML_GLOBAL_PATH +
               "\\tools\\visualizers\\real_function_visualizer\\MML_RealFunctionVisualizer.exe";
static const std::string MML_PATH_SurfaceViz  = MML_GLOBAL_PATH +
               "\\tools\\visualizers\\scalar_function_2d_visualizer\\MML_ScalarFunction2Visualizer.exe";
static const std::string MML_PATH_ParametricCurveViz = MML_GLOBAL_PATH +
               "\\tools\\visualizers\\parametric_curve_visualizer\\MML_ParametricCurveVisualizer.exe";
static const std::string MML_PATH_VectorFieldViz = MML_GLOBAL_PATH +
               "\\tools\\visualizers\\vector_field_visualizer\\MML_VectorFieldVisualizer.exe";
```

We have a Visualizer class with following set of visualization capabilities

```
class Visualizer
{
  static inline std::string _pathResultFiles{ MML_PATH_ResultFiles };

  static inline std::string _pathRealFuncViz{ MML_PATH_RealFuncViz };
  static inline std::string _pathSurfaceViz{ MML_PATH_SurfaceViz };
  static inline std::string _pathParametricCurveViz{ MML_PATH_ParametricCurveViz };
  static inline std::string _pathVectorFieldViz{ MML_PATH_VectorFieldViz };

public:
  static void VisualizeRealFunction(const IRealFunction& f, std::string title, Real x1, Real x2,
                                    int numPoints, std::string fileName) { ... }

  static void VisualizeMultiRealFunction(std::vector<IRealFunction*> funcs, std::string title,
                                         Real x1, Real x2, int numPoints, std::string fileName) { ... }

  static void VisualizeScalarFunc2DCartesian(const IScalarFunction<2>& func, std::string title,
                                             Real x1, Real x2, int numPointsX,
                                             Real y1, Real y2, int numPointsY, std::string fileName) { ... }

  static void VisualizeVectorField3DCartesian(const IVectorFunction<3>& func, std::string title,
                                              Real x1, Real x2, int numPointsX,
                                              Real y1, Real y2, int numPointsY,
                                              Real z1, Real z2, int numPointsZ, std::string fileName) { ... }

  static void VisualizeParamCurve3D(const IRealToVectorFunction<3>& f, std::string title, Real t1, Real t2,
                                    int numPoints, std::string fileName) { ... }

  static void VisualizeMultiParamCurve3D(std::vector<std::string> fileNames) { ... }

  static void VisualizeODESysSolAsMultiFunc(const ODESystemSolution& sol,
                                            std::string title, std::string fileName) { ... }

  static void VisualizeODESysSolAsParamCurve3(const ODESystemSolution& sol,
                                              std::string title, std::string fileName) { ... }
};
```

Implementations are very much alike and follow similar pattern

```
    static void VisualizeRealFunction(const IRealFunction& f, std::string title,
                                      Real x1, Real x2, int numPoints, std::string fileName)
    {
      std::string name = _pathResultFiles + fileName;
      Serializer::SaveRealFuncEquallySpacedDetailed(f, title, x1, x2, numPoints, name);

#if defined(MML_PLATFORM_WINDOWS)
      std::string command = _pathRealFuncViz + " " + name;
      system(command.c_str());
#else
      std::cout << "VisualizeRealFunction: Not implemented for this OS" << std::endl;
#endif
    }
```

4. From colliding mechanical balls to gas laws

Embarking on our journey with some 17th and 18th century physics.

Tasks at hand:

- Given positions and initial velocities of N simple mechanical balls, confined to 2D or 3D container, calculate their positions in time.
- Relate to gas laws

PHYSICS – COLLIDING MECHANICAL BALLS

Starting with basics of physics

We'll need First newton law, and law of conservation of energy together with law of conservation of momentum.

One-dimensional case

TODO - Two balls, pictured on a line.

Conservation of momentum:

$$m_A v_{A1} + m_B v_{B1} = m_A v_{A2} + m_B v_{B2}.$$

Conservation of energy:

$$\frac{1}{2} m_A v_{A1}^2 + \frac{1}{2} m_B v_{B1}^2 = \frac{1}{2} m_A v_{A2}^2 + \frac{1}{2} m_B v_{B2}^2.$$

When solved, gives:

$$\begin{aligned} v_{A2} &= \frac{m_A - m_B}{m_A + m_B} v_{A1} + \frac{2m_B}{m_A + m_B} v_{B1} \\ v_{B2} &= \frac{2m_A}{m_A + m_B} v_{A1} + \frac{m_B - m_A}{m_A + m_B} v_{B1}. \end{aligned}$$

Center of mass frame doesn't change!

Two-dimensional case

Dependent on the point of collision, and after lengthy calculation, in an angle-free representation, the changed velocities are computed using the centers $\mathbf{x}_1$ and $\mathbf{x}_2$ at the time of contact as:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2),$$
$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

Based on these equations, we can easily implement calculations of velocity vectors for two balls after collision:

```
// https://en.wikipedia.org/wiki/Elastic_collision - calculating new velocities after collision
double   m1 = ball1.Mass(), m2 = ball2.Mass();

Vec2Cart v1(ball1.V()), v2(ball2.V());

Vec2Cart v1_v2(v1 - v2);
Vec2Cart x1_x2(x2, x1);

Vec2Cart v1_new = v1 - 2 * m2 / (m1 + m2) * (v1_v2 * x1_x2) / POW2(x1_x2.NormL2()) * Vec2Cart(x2, x1);
Vec2Cart v2_new = v2 - 2 * m1 / (m1 + m2) * (v1_v2 * x1_x2) / POW2(x1_x2.NormL2()) * Vec2Cart(x1, x2);

ball1.V() = v1_new;
ball2.V() = v2_new;

// adjusting new ball positions
ball1.Pos() = x1 + ball1.V() * (dt - tCollision);
ball2.Pos() = x2 + ball2.V() * (dt - tCollision);
```

DOING A 2D SIMULATION - COLLISIONSIMULATOR2D

Building a simple collision simulator.

https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_04_collision_simulator/collision_simulator_2d.cpp

Starting with a model of 2D ball.

```
struct Ball2D
{
private:
  double _mass;
  double _radius;
  Point2Cartesian _position;
  Vector2Cartesian _velocity;

public:
  Ball2D(double mass, double radius, const Point2Cartesian& position,
         const Vector2Cartesian& velocity) { ... }

  double  Mass() const { return _mass; }
  double& Mass() { return _mass; }

  double  Rad() const { return _radius; }
  double& Rad() { return _radius; }

  Point2Cartesian  Pos() const { return _position; }
  Point2Cartesian& Pos() { return _position; }

  Vector2Cartesian  V() const { return _velocity; }
  Vector2Cartesian& V() { return _velocity; }
};
```

Container for our balls.

Simple rectangle, with bottom left corner at point (0,0), with given width and height.

```
struct Container2D
{
  double _width;
  double _height;

  std::vector<Ball2D> _balls;

  Container2D() : _width(1000), _height(1000) {}
  Container2D(double width, double height) : _width(width), _height(height) {}

  // add body
  void AddBall(const Ball2D& body) { ... }

  // get body
  Ball2D& Ball(int i) { ... }

  // check if ball is out of bounds and handle it
  void CheckAndHandleOutOfBounds(int ballIndex) { ... }
};
```

Where CheckAndHandleOutOfBounds() function is crucial:

```
void CheckAndHandleOutOfBounds(int ballIndex)
{
  const Ball2D& ball = ball;

  // left wall collision
  if (ball.Pos().X() < ball.Rad() && ball.V().X() < 0)
  {
    ball.Pos().X() = ball.Rad() + (ball.Rad() - ball.Pos().X()); // Get back to box!
    ball.V().X() *= -1;
  }
  // right wall collision
  if (ball.Pos().X() > _width - ball.Rad() && ball.V().X() > 0)
  {
    ball.Pos().X() -= (ball.Pos().X() + ball.Rad()) - _width;
    ball.V().X() *= -1;
  }
  // bottom wall collision
  if (ball.Pos().Y() < ball.Rad() && ball.V().Y() < 0)
  {
    ball.Pos().Y() = ball.Rad() + (ball.Rad() - ball.Pos().Y());
    ball.V().Y() *= -1;
  }
  // top wall collision
  if (ball.Pos().Y() > _height - ball.Rad() && ball.V().Y() > 0)
  {
    ball.Pos().Y() -= (ball.Pos().Y() + ball.Rad()) - _height;
    ball.V().Y() *= -1;
  }
}
```

And finally, CollisionSimulator2D class, implementing simulator.

```
class CollisionSimulator2D
{
  Container2D _box;

public:
  CollisionSimulator2D() { }
  CollisionSimulator2D(const Container2D& box) : _box(box) {}

  double DistBalls(int i, int j){ ... }

  bool HasBallsCollided(int i, int j){ ... }

  void SimulateOneStep(double dt)
  {
    // ...
    int NumBalls = _box._balls.size();

    // first, update all ball's positions, and handle out of bounds
    for (int i = 0; i < NumBalls; i++)
    {
      _box._balls[i].Pos() = _box._balls[i].Pos() + _box._balls[i].V() * dt;

      _box.CheckAndHandleOutOfBounds(i);
    }

    // check for collisions, and handle it if there is one
    for (int m = 0; m < NumBalls - 1; m++) {
      for (int n = m + 1; n < NumBalls; n++)
      {
        if (HasBallsCollided(m, n)){ ... }
      }
    }
  }
};
```

Visualizing container with balls with Qt

TODO

3D SIMULATION

Implementation is the same, only using Vector3Cartesian!

Visualizing with WPF?

PHYSICS – IDEAL GAS

Model of colliding balls in elastic collision is fairly good approximation for ideal gas.

Little bit of formulas.

SIMULATING IDEAL GAS

Developed collision simulator can be used to verify basic gas laws, as our mechanical balls model represents most of the ideal gas approximations.

Calculating pressure by summing momentum transferred to walls during collisions

Calculating temperature from average kinetic energy of balls

Verifying $p V = n R T$ gas equation

PISTON SIMULATION

Cylinder with movable wall in the middle

What happens if we inject highly energetic balls into one side?

Could we simulate chemical reactions?

- When different types of “molecules” collide, there is released energy, which translates to higher velocities after “collision”

IMPROVING EFFICIENCY FOR LARGE NUMBER OF BALLS

Implemented algorithm scales as $O(N^2)$!

Feasible for under 1000 balls.

What can we do if we want to simulate one million balls?

5. Pendulum - Newton laws and ODE solvers

Introducing forces … and those kings of numerical simulation, ODE solvers.

Tasks at hand:

- Introduce pendulum as one of the simplest physical systems
- Introduce numerical ODE solvers and calculate exact motion of pendulum
- Compare obtained solutions with analytical one
- What if we add air resistance?

PENDULUM PHYSICS

TODO – intro to pendulum

Applying Newton second law to tangential component, we get

$$F = -mg\sin\theta = ma,$$
$$a = -g\sin\theta,$$

The negative sign on the right-hand side means that θ and a always point in opposite directions

This linear acceleration a along the red axis can be related to the change in angle θ by the arc length formulas; s is arc length:

$$s = \ell\theta,$$
$$v = \frac{ds}{dt} = \ell\frac{d\theta}{dt},$$
$$a = \frac{d^2s}{dt^2} = \ell\frac{d^2\theta}{dt^2},$$

Giving our equation of motion

$$\ell\frac{d^2\theta}{dt^2} = -g\sin\theta,$$
$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\sin\theta = 0.$$

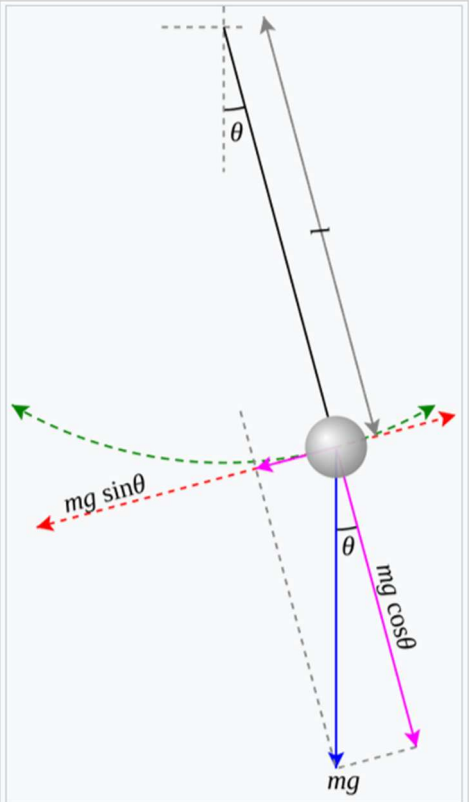


Figure 1. Force diagram of a simple gravity pendulum.

Picture from Wikipedia.

Exact equation of motion

Linear approximation for small angles

How can we calculate exact values?

SOLVING ODES NUMERICALLY

Euler method

Midpoint method

Runge-Kutta methods

Adaptive step size methods

IMPLEMENTING ODE SOLVERS IN C++

ODESystem class

Basic interface, used for modelling system of ordinary differential equations.

```
class IODESystem
{
public:
  virtual int   getDim() const = 0;
  virtual void  derivs(const Real t, const Vector<Real> &x, Vector<Real> &dxdt) const = 0;
};
```

You can use it to create your own ODE system, by inheriting this interface in your class and providing two virtual methods.

TODO – example ODE from introduction, but for now, Pendulum example from chapter 5 will do.

```
class PendulumODE : public IODESystem
{
  Real _Length;
public:
  PendulumODE(Real length) : _Length(length) {}

  int  getDim() const override { return 2; }
  void derivs(const Real t, const MML::Vector<Real> &x, MML::Vector<Real> &dxdt) const override
  {
    dxdt[0] = x[1];
    dxdt[1] = -9.81 / _Length * sin(x[0]);
  }
};
```

If you already have a (static) function ready with derivatives calculation, or maybe your system equations can be easily coded with simple lambda, you can use provided ODESystem class.

```
class ODESystem : public IODESystem
{
protected:
  int _dim;
  void (*_func)(Real, const Vector<Real>&, Vector<Real>&);

public:
  ODESystem() : _dim(0), _func(nullptr) { }
  ODESystem(int n, void (*inFunc)(Real, const Vector<Real>&, Vector<Real>&))
    : _dim(n), _func(inFunc) { }

  int getDim() const { return _dim; }
  void derivs(const Real x, const Vector<Real>& y, Vector<Real>& dydx) const { ... }
};
```

Example of pendulum system definition using lambda.

```
void Demo_ExactPendulum()
{
  ODESystem pendSys(2, [](Real t, const Vector<Real>& x, Vector<Real>& dxdt)
    {
      Real Length = 1.0;
      dxdt[0] = x[1];
      dxdt[1] = -9.81 / Length * sin(x[0]);
    });
```

Created object pendSys is ready to be supplied wherever IODESystem reference is expected.

Important thing to note is that in case with lambda definition, Length parameter is hard-coded within lambda, and can't be changed in run-time!

ODESystemSolution class

For storing our calculation results.

```
class ODESystemSolution
{
  int  _numStepsOK, _numStepsBad;

public:
  int  _sys_dim;
  int  _totalSavedSteps;

  Real _t1, _t2;
  Vector<Real> _tval;
  Matrix<Real> _xval;

  ODESystemSolution(Real x1, Real x2, int dim) { ... }
  ODESystemSolution(Real x1, Real x2, int dim, int maxSteps) { ... }

  void incrementNumStepsOK()  { _numStepsOK++; }
  void incrementNumStepsBad() { _numStepsBad++; }

  int getNumStepsOK() const { return _numStepsOK; }
  int getNumStepsBad() const { return _numStepsBad; }
  int getTotalNumSteps() const { return _numStepsOK + _numStepsBad; }

  Vector<Real> getXValues() const { return _tval; }
  Matrix<Real> getYValues() const { return _xval; }
  Vector<Real> getYValues(int component) const { ... }


  void fillValues(int ind, Real x, Vector<Real>& y) { ... }
  void setFinalSize(int numDoneSteps) { ... }

  LinearInterpRealFunc  getSolutionAsLinearInterp(int component) const { ... }
  PolynomInterpRealFunc getSolutionAsPolyInterp(int component, int polyOrder) const
  SplineInterpRealFunc  getSolutionAsSplineInterp(int component) const { ... }
```

Step calculators

Have interface defined in IODESystemStelClaculators.h

```
class IODESystemStepCalculator
{
public:
  virtual void calcStep(const IODESystem& odeSystem,
                        const Real x, const Vector<Real>& y_start, const Vector<Real>& dydx,
                        const Real h, Vector<Real>& yout, Vector<Real>& yerr) const = 0;
};
```

Basic RK4

```
class RK4_FixedStep_Calculator : public IODESystemStepCalculator
{
public:
  // For a given ODESystem, of dimension n, and given values for the variables y_start[0..n-1]
  // and their derivatives dydx[0..n-1] known at x, uses the fourth-order Runge-Kutta method
  // to advance the solution over an interval h and return the incremented variables as yout[0..n-1].
  void calcStep(const IODESystem& odeSystem,
                const Real t, const Vector<Real> &x_start, const Vector<Real> &dxdt,
                const Real h, Vector<Real> &x_out, Vector<Real> &x_err_out) const override
  {
    int i, n = odeSystem.getDim();
    Vector<Real> dx_mid(n), dx_temp(n), x_temp(n);

    Real xh, hh, h6;
    hh = h * 0.5;
    h6 = h / 6.0;
    xh = t + hh;

    for (i = 0; i < n; i++)                       // First step
      x_temp[i] = x_start[i] + hh * dxdt[i];

    odeSystem.derivs(xh, x_temp, dx_temp);        // Second step

    for (i = 0; i < n; i++)
      x_temp[i] = x_start[i] + hh * dx_temp[i];

    odeSystem.derivs(xh, x_temp, dx_mid);         // Third step

    for (i = 0; i < n; i++) {
      x_temp[i] = x_start[i] + h * dx_mid[i];
      dx_mid[i] += dx_temp[i];
    }

    odeSystem.derivs(t + h, x_temp, dx_temp);     // Fourth step

    for (i = 0; i < n; i++)
      x_out[i] = x_start[i] + h6 * (dxdt[i] + dx_temp[i] + 2.0 * dx_mid[i]);
  }
};
```

Fixed-step ODE solver

```
class ODESystemFixedStepSolver
{
  IODESystem& _odeSys;
  IODESystemStepCalculator& _stepCalc;

public:
  ODESystemFixedStepSolver(IODESystem& inOdeSys, IODESystemStepCalculator& inStepCalc)
    : _odeSys(inOdeSys), _stepCalc(inStepCalc) { }

  // For given numSteps, ODESystemSolution will have numSteps+1 values!
  ODESystemSolution integrate(const Vector<Real>& initCond, Real x1, Real x2, int numSteps)
  {
    int   dim = _odeSys.getDim();

    ODESystemSolution sol(x1, x2, dim, numSteps);
    Vector<Real>      y(initCond), y_out(dim), dydx(dim), y_err(dim);

    sol.fillValues(0, x1, y);

    Real x = x1;
    Real h = (x2 - x1) / numSteps;

    for (int k = 1; k <= numSteps; k++) {
      _odeSys.derivs(x, y, dydx);

      _stepCalc.calcStep(_odeSys, x, y, dydx, h, y_out, y_err);

      x += h;

      y = y_out;
      sol.fillValues(k, x, y);
    }

    return sol;
  }
};
```

Implementing adaptive step ODE solver

Base stepper

```
class StepperBase {
public:
  // references that stepper gets from the solver
  IODESystem&   _sys;

  Real&         _t;
  Vector<Real>& _x;
  Vector<Real>& _dxdt;

  Real  _absTol, _relTol;
  Real  _eps;

  Real  _tOld;

  Real  _hDone, _hNext;

  Vector<Real> _xout, _xerr;

public:
  StepperBase(IODESystem& sys, Real& t, Vector<Real>& x, Vector<Real>& dxdt)
    : _sys(sys), _t(t), _x(x), _dxdt(dxdt) {
  }

  virtual void doStep(Real htry, Real eps) = 0;

  void setHDone(Real h) { _hDone = h; }
  void setHNext(Real h) { _hNext = h; }
};
```

Runge-Kutta 4th order Cash-Karp adaptive size algorithm

Example of simple adaptive size algorithm

```
class RK4_CashKarp_Stepper : public StepperBase
{
private:
  RK4_CashKarp_Calculator _stepCalc;

public:
  RK4_CashKarp_Stepper(IODESystem& sys, Real& t, Vector<Real>& x, Vector<Real>& dxdt)
    : StepperBase(sys, t, x, dxdt) { }

  /*
  Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and
  adjust stepsize. Input are the dependent variable vector y[1.._numPoints] and its derivative dydx[1.._numPoints]
  at the starting value of the independent variable x. Also input are the stepsize to be attempted
  htry, the required accuracy eps,
  On output, y and x are replaced by their new values, hdid is the stepsize that was
  actually accomplished, and hnext is the estimated next stepsize. derivs is the user-supplied
  routine that computes the right-hand side derivatives.
  */
  void doStep(Real htry, Real eps) override { ... }
};
```

Adaptive step calculation

```
void doStep(Real htry, Real eps) override
{
  const Real SAFETY = 0.9, PGROW = -0.2, PSHRNK = -0.25, ERRCON = 1.89e-4;
  Real  errmax, h, htemp, tnew;
  int   i, n = _sys.getDim();
  Vector<Real> xerr(n), xscale(n), xtemp(n);

  h = htry;           // Set stepsize to the initial trial value.

  // Scaling used to monitor accuracy.
  for (i = 0; i < n; i++)
    xscale[i] = fabs(_x[i]) + fabs(_dxdt[i] * h) + 1e-30;

  for (;;) {
    _stepCalc.calcStep(_sys, _t, _x, _dxdt, h, xtemp, xerr);

    errmax = 0.0;               // Evaluating accuracy.
    for (int i = 0; i < n; i++)
      errmax = std::max(errmax, fabs(xerr[i] / xscale[i]));
    errmax /= eps;              // Scale relative to required tolerance
    if (errmax <= 1.0)
      break;                    // Step succeeded. Compute size of next step.

    htemp = SAFETY * h * pow(errmax, PSHRNK);
    // Truncation error too large, reduce stepsize, but no more than a factor of 10.
    if (h >= Real{ 0 })
      h = std::max(htemp, 0.1 * h);
    else
      h = std::min(htemp, 0.1 * h);

    tnew = _t + h;

    if (tnew == _t) throw("stepsize underflow in rkqs");
  }
  // computing size of the next step
  if (errmax > ERRCON)
    _hNext = SAFETY * h * pow(errmax, PGROW);
  else
    _hNext = 5.0 * h;       // No more than a factor of 5 increase

  _hDone = h;
  _t += h;

  for (i = 0; i < n; i++)
    _x[i] = xtemp[i];
}
```

ODESolver class as master integrator

- Templated on stepper
- Gets ODESystem as input
- Produces ODESystemSolution

```
template<class Stepper> class ODESystemSolver
{
  IODESystem&   _sys;
  Stepper       _stepper;

  // used as reference by stepper!
  Real          _curr_t;
  Vector<Real>  _curr_x;
  Vector<Real>  _curr_dxdt;

public:
  ODESystemSolver(IODESystem& sys)
    : _sys(sys), _stepper(sys, _curr_t, _curr_x, _curr_dxdt)
  {
    _curr_x.Resize(_sys.getDim());
    _curr_dxdt.Resize(_sys.getDim());
  }

  int getDim() { return _sys.getDim(); }

  // ODE system integrator using given stepper for adaptive stepsize control.
  // Integrates starting values ystart[1..nvar] from t1 to t2 with accuracy eps
  ODESystemSolution integrate(const Vector<Real>& initCond,
                              Real t1, Real t2, Real minSaveInterval,
                              Real eps, Real h1, Real hmin = 0) { ... }
};
```

Integrate function

```
// ODE system integrator using given stepper for adaptive stepsize control.
// Integrates starting values ystart[1..nvar] from t1 to t2 with accuracy eps
ODESystemSolution integrate(const Vector<Real>& initCond,
                            Real t1, Real t2, Real minSaveInterval,
                            Real eps, Real h1, Real hmin = 0)
{
  Real xsav, h;
  int i, stepNum, numSavedSteps=0, sysDim = _sys.getDim();
  int expectedSteps = (int)((t2 - t1) / h1) + 1;
  ODESystemSolution sol(t1, t2, sysDim, expectedSteps);

  _curr_t = t1;
  _curr_x = initCond;
  h = SIGN(h1, t2 - t1);

  if (expectedSteps > 0)  xsav = _curr_t - minSaveInterval * 2.0;

  for (stepNum = 0; stepNum < Defaults::ODESolverMaxSteps; stepNum++) {
    _sys.derivs(_curr_t, _curr_x, _curr_dxdt);

    // storing intermediate results, if we have advanced enough
    if (expectedSteps > 0 && fabs(_curr_t - xsav) > fabs(minSaveInterval))
    {
      sol.fillValues(numSavedSteps, _curr_t, _curr_x);
      numSavedSteps++;
      xsav = _curr_t;
    }
    // If stepsize overshoots end of interval, decrease to adjust
    if ((_curr_t + h - t2) * (_curr_t + h - t1) > 0.0)
      h = t2 - _curr_t;

    _stepper.doStep(h, eps);

    if (_stepper._hDone == h)
      sol.incrementNumStepsOK();
    else
      sol.incrementNumStepsBad();

    // check if we are done
    if ((_curr_t - t2) * (t2 - t1) >= 0.0) {
      if (expectedSteps != 0) {
        sol.fillValues(numSavedSteps, _curr_t, _curr_x);
      }
      sol.setFinalSize(numSavedSteps);
      return sol;
    }
    if (fabs(_stepper._hNext) <= hmin) throw ODESolverError("Step size too small in integrate");

    h = _stepper._hNext;
  }
  throw ODESolverError("Too many steps in routine integrate");
}
```

SOLVING PENDULUM

Defining ODESystem class representing pendulum

```
class PendulumODE : public IODESystem
{
  Real _Length;
public:
  PendulumODE(Real length) : _Length(length) {}

  int  getDim() const override { return 2; }
  void derivs(const Real x, const MML::Vector<Real>& y, MML::Vector<Real>& dydx) const override
  {
    dydx[0] = y[1];
    dydx[1] = -9.81 / _Length * sin(y[0]);
  }
};
```

Solving pendulum with our RK4 ODE solvers

Using fixed step-size and adaptive step-size RK solvers to solve it.

```
Real  pendulumLen = 1.0;
PendulumODE   sys = PendulumODE(pendulumLen);

Real  t1 = 0.0, t2 = 10.0;
int   expectNumSteps = 100;
Real  minSaveInterval = (t2 - t1) / expectNumSteps;
Real  initAngle = 0.5;
Vector<Real>  initCond{ initAngle, 0.0 };

ODESystemFixedStepSolver  fixedSolver(pendSys, StepCalculators::RK4_Basic);
ODESystemSolution solFixed = fixedSolver.integrate(initCond, t1, t2, expectNumSteps);

ODESystemSolver<RK4_CashKarp_Stepper> adaptSolver(sys);
ODESystemSolution solAdapt = adaptSolver.integrate(initCond, t1, t2, minSaveInterval / 1.21, 1e-06, 0.01);

Vector<Real> x_fixed  = solFixed.getXValues();
Vector<Real> y1_fixed = solFixed.getYValues(0);
Vector<Real> y2_fixed = solFixed.getYValues(1);

Vector<Real> x_adapt  = solAdapt.getXValues();
Vector<Real> y1_adapt = solAdapt.getYValues(0);
Vector<Real> y2_adapt = solAdapt.getYValues(1);

std::cout << "\n\n****  Runge-Kutta 4th order - fixed stepsize  **********  Runge-Kutta 4th order - adaptive stepper  ****\n";
std::vector<ColDesc>        vecNames{ ColDesc("t", 11, 2, 'F'), ColDesc("angle", 15, 8, 'F'), ColDesc("ang.vel.", 15, 8, 'F'),
                                      ColDesc("t", 22, 2, 'F'), ColDesc("angle", 12, 8, 'F'), ColDesc("ang.vel.", 12, 8, 'F') };
std::vector<Vector<Real>*>  vecVals{ &x_fixed, &y1_fixed, &y2_fixed,
                                     &x_adapt, &y1_adapt, &y2_adapt };
VerticalVectorPrinter vvp(vecNames, vecVals);

vvp.Print();
```

Console output

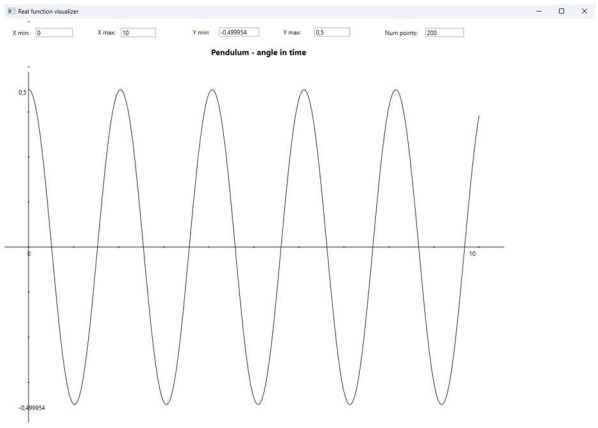
** Runge-Kutta 4th order - fixed step-size **			** Runge-Kutta 4th order - adaptive stepper **		
t	angle	ang.vel.	t	angle	ang.vel.
0.00	0.50000000	0.00000000	0.00	0.50000000	0.00000000
0.10	0.47665342	-0.46353601	0.13	0.45778475	-0.61776411
0.20	0.40863848	-0.88674198	0.22	0.38831757	-0.97008776
0.30	0.30196570	-1.23045133	0.32	0.28138501	-1.27679705
0.40	0.16638709	-1.45971284	0.42	0.14224787	-1.48447498
0.50	0.01472426	-1.54905478	0.51	0.00100032	-1.54978139
0.60	-0.13836619	-1.48800001	0.66	-0.22942350	-1.37395149
0.70	-0.27806011	-1.28374679	0.75	-0.33679056	-1.13999272
0.80	-0.39107639	-0.95930119	0.84	-0.42643042	-0.80300872
0.90	-0.46701295	-0.54826751	0.94	-0.48458761	-0.37805645
1.00	-0.49912094	-0.08932350	1.04	-0.49862378	0.11372398
…					
8.40	0.36153984	-1.06352601	8.58	0.12012427	-1.50349002
8.50	0.23961489	-1.35605279	8.67	-0.01867399	-1.54869254
8.60	0.09463244	-1.52040364	8.83	-0.25118286	-1.33647917
8.70	-0.05952356	-1.53773695	8.92	-0.35626270	-1.08160447
8.80	-0.20790146	-1.40600366	9.01	-0.44074004	-0.72588472
8.90	-0.33622641	-1.14048435	9.11	-0.49123664	-0.28602377
9.00	-0.43247253	-0.77000296	9.20	-0.49771419	0.14661937
9.10	-0.48793268	-0.33118027	9.29	-0.46428821	0.57000767
9.20	-0.49774494	0.13629050	9.38	-0.39991174	0.92399290
9.30	-0.46106272	0.59202240	9.47	-0.29786175	1.24013268
9.40	-0.38107371	0.99586669	9.57	-0.16254622	1.46398440
9.50	-0.26491163	1.30947278	9.68	-0.00443368	1.54973588
9.60	-0.12330802	1.50005315	9.83	0.22803362	1.37621501
9.70	0.03023798	1.54600550	9.92	0.33552554	1.14359632
9.80	0.18084654	1.44190607	10.00	0.41786505	0.84481904
9.90	0.31398298	1.19992327			
10.00	0.41709503	0.84673507			

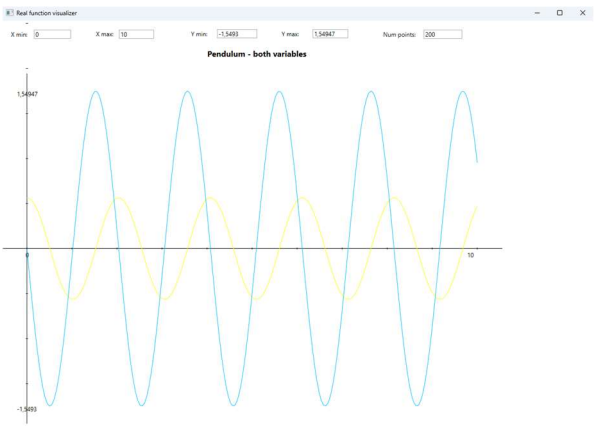
Visualizing solutions graphically

Extracting RealFunction from our discretized ODE solutions.

```
// getting solutions as polynomials
PolynomInterpRealFunc    solFixedPolyInterp0 = solFixed.getSolutionAsPolyInterp(0, 3);
PolynomInterpRealFunc    solFixedPolyInterp1 = solFixed.getSolutionAsPolyInterp(1, 3);

PolynomInterpRealFunc    solAdaptPolyInterp0 = solAdapt.getSolutionAsPolyInterp(0, 3);
PolynomInterpRealFunc    solAdaptPolyInterp1 = solAdapt.getSolutionAsPolyInterp(1, 3);

SplineInterpRealFunc     solSplineInterp0 = solAdapt.getSolutionAsSplineInterp(0);
SplineInterpParametricCurve<2> spline(solAdapt.getYValues());

Visualizer::VisualizeRealFunction(solAdaptPolyInterp0, "Pendulum - angle in time",
                                  0.0, 10.0, 200, "pendulum_angle.txt");

// shown together
Visualizer::VisualizeMultiRealFunction({ &solAdaptPolyInterp0, &solAdaptPolyInterp1 },
                                       "Pendulum - both variables", 0.0, 10.0, 200,
                                       "pendulum_multi_real_func.txt");
```

Vizualizations with vizualizers.





Calculating pendulum period

We are interested in the period of our pendulum, and we can obtain that from solution by analyzing roots of solution functions – checking where they cross abscissa, and looking at distance between neighbor crossing.

```
Real calcFunctionPeriod(const IRealFunction &func, Real t1, Real t2)
{
  int   numFoundRoots;
  Vector<Real> root_brack_x1(10), root_brack_x2(10);
  FindRootBrackets(func, t1, t2, 4000, root_brack_x1, root_brack_x2, numFoundRoots);

  Vector<Real> roots(numFoundRoots);
  Vector<Real> rootDiffs(numFoundRoots - 1);
  for (int i = 0; i < numFoundRoots; i++)
  {
    roots[i] = FindRootBisection(func, root_brack_x1[i], root_brack_x2[i], 1e-7);

    if( i>0 )
      rootDiffs[i - 1] = roots[i] - roots[i - 1];
  }

  return Statistics::Avg(rootDiffs);
}
```

We can calculate exact period, based on analytical solution

```
double calculatePendulumPeriod(double length, double initialAngle)
{
  const double g = 9.81;
  double k = std::sin(initialAngle / 2);
  double ellipticIntegral = std::comp_ellint_1(k);

  double period = 4 * std::sqrt(length / g) * ellipticIntegral;

  return period;
}
```

And compare results

```
Real periodLinear = 2.0 * Constants::PI * sqrt(pendulumLen / 9.81);
std::cout << "Pendulum period approx. linear : " << periodLinear << std::endl;

Real simulPeriodFixed = calcFunctionPeriod(solFixedPolyInterp0, t1, t2);
std::cout << "Pendulum period RK4 fixed step : " << 2 * simulPeriodFixed << std::endl;

Real simulPeriodAdapt = calcFunctionPeriod(solAdaptPolyInterp0, t1, t2);
std::cout << "Pendulum period RK4 adapt.step : " << 2 * simulPeriodAdapt << std::endl;

// calculate exact period for pendulum of length L, and given initial angle phi
Real exactPeriod = calculatePendulumPeriod(pendulumLen, initAngle);
std::cout << "Pendulum period analytic exact : " << exactPeriod << std::endl;
```

Pendulum period approx. linear : 2.0060667

Pendulum period RK4 fixed step : 2.0379898

Pendulum period RK4 adapt.step: 2.0378686

Pendulum period analytic exact : 2.0378679

Investigating dependence on initial angle

How does the pendulum period changes, depending on initial angle (ie, how much was pendulum deflected from equilibrium position).

Code

```
Vector<Real> initAnglesDeg{ 1, 2, 5, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0 };
Vector<Real> initAngles(initAnglesDeg.size());        // convert to radians
for (int i = 0; i < initAngles.size(); i++)
  initAngles[i] = initAnglesDeg[i] * Constants::PI / 180.0;

Vector<Real> periods(initAngles.size());
Real periodLin, periodSimulFixed, periodSimulAdapt, periodExact;

std::cout << "\nAngle      Linear.      Exact     Fix.step.sim Adapt.step.sim" << std::endl;
for (int i = 0; i < initAngles.size(); i++)
{
  initCond[0] = initAngles[i];

  // calculate period from fixed solution
  ODESystemSolution     solF = fixedSolver.integrate(initCond, t1, t2, expectNumSteps);
  PolynomInterpRealFunc solFInterp = solF.getSolutionAsPolyInterp(0, 3);
  periodSimulFixed = calcFunctionPeriod(solFInterp, t1, t2);

  // calculate period from adaptive solution
  ODESystemSolution     solA = adaptSolver.integrate(initCond, t1, t2, minSaveInterval, 1e-06, 0.01);
  PolynomInterpRealFunc solAInterp = solA.getSolutionAsPolyInterp(0, 3);
  periodSimulAdapt = calcFunctionPeriod(solAInterp, t1, t2);

  // analytical formulas
  periodLin = 2.0 * Constants::PI * sqrt(pendulumLen / 9.81);
  periodExact = calculatePendulumPeriod(pendulumLen, initAngles[i]);

  std::cout << std::setw(2) << initAnglesDeg[i] << " deg:  " << periodLin << "    "
            << periodExact << "    " << 2 * periodSimulFixed << "    " << 2 * periodSimulAdapt << std::endl;
}
```

Angle	Linear.	Exact	Fix.step.sim	Adapt.step.sim
1 deg:	2.0060667	2.0061049	2.0062607	2.0061015
2 deg:	2.0060667	2.0062195	2.0063751	2.0062195
5 deg:	2.0060667	2.0070219	2.0071766	2.0070184
10 deg:	2.0060667	2.0098926	2.0100438	2.0098799
20 deg:	2.0060667	2.0214513	2.0215905	2.0214573
30 deg:	2.0060667	2.0409899	2.0411092	2.0409933
40 deg:	2.0060667	2.0689379	2.0690328	2.0689287
50 deg:	2.0060667	2.1059346	2.1060029	2.1059113
60 deg:	2.0060667	2.1528747	2.152909	2.1528623
70 deg:	2.0060667	2.2109762	2.2109703	2.2109792
80 deg:	2.0060667	2.2818859	2.2818311	2.2818973

Dependence of number of steps on EPS

Code

```
Vector<Real> acc{ 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8 };
Vector<Real> numSteps(acc.size());

std::cout << "\nAccuracy   Num steps   OK steps   Bad steps" << std::endl;
for (int i = 0; i < acc.size(); i++)
{
  ODESystemSolver<RK4_CashKarp_Stepper> adaptSolver(sys);
  ODESystemSolution solAdapt = adaptSolver.integrate(initCond, t1, t2, minSaveInterval, acc[i], 0.01);
  numSteps[i] = solAdapt.getTotalNumSteps();

  std::cout << std::setw(7) << acc[i] << "        " << std::setw(3) << numSteps[i] << "        "
            << std::setw(3) << solAdapt.getNumStepsOK() << "        "
            << std::setw(3) << solAdapt.getNumStepsBad() << std::endl;
}
```

Results

```
Accuracy   Num steps   OK steps   Bad steps
   0.01          17         13           4
  0.001          26         17           9
 0.0001          40         30          10
  1e-05          61         57           4
  1e-06          96         91           5
  1e-07         157        143          14
  1e-08         255        241          14
```

LONG TERM SIMULATION

Need for symplectic integrators, that preserve phase space volume

ADDING AIR RESISTANCE

Basically, it is a dumped oscillating system

6. Spherical and double pendulum – working with Lagrangians

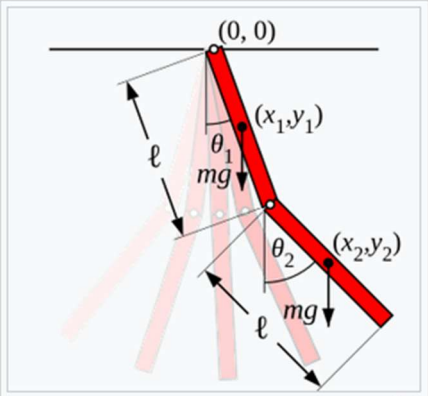
Going all-in with Lagrangian formulation of classical mechanics.

Tasks at hand:

- Introduce basics of Lagrangian formulation of classical mechanics
- Spherical pendulum as an example

PHYSICS – LAGRANGIAN FORMULATION IN CLASSICAL MECHANICS

PHYSICS - DOUBLE PENDULUM



Double compound pendulum

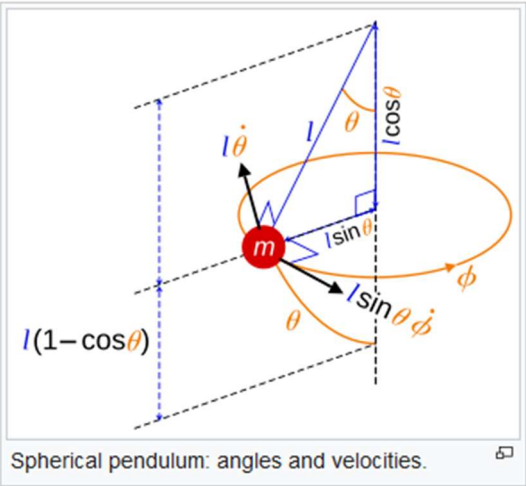
After lengthy analysis, we get equation of motion in theta1 and theta2.

$$\tfrac{1}{3}\ell\ddot{\theta}_2 + \tfrac{1}{2}\ell\ddot{\theta}_1\cos(\theta_1-\theta_2) - \tfrac{1}{2}\ell\dot{\theta}_1^2\sin(\theta_1-\theta_2) + \tfrac{1}{2}g\sin\theta_2 = 0.$$

Which, when simplified to system of first order differential equations gives this:

TODO!!!

SPHERICAL PENDULUM



Spherical pendulum: angles and velocities.

Equation of motion for theta angle

$$\ddot{\theta} = \sin\theta\cos\theta\dot{\phi}^2 - \frac{g}{l}\sin\theta$$

$$\ddot{\phi}\,\sin\theta = -2\,\dot{\theta}\,\dot{\phi}\,\cos\theta.$$

7. Hitting a ball with a baseball bat

Working it all the way for real situation.

Tasks at hand:

- What happens when we hit a baseball with a baseball bat?
- Adding air resistance
- Adding effect of drag
- Adding effect of spin

VACUUM SOLUTION

AIR RESISTANCE

EFFECT OF BASEBALL PROPERTIES - DRAG

EFFECT OF SPIN

8. Central potential – gravity field

Investigating that all-present force in our lives … gravity.

Tasks at hand:

- Verify Kepler laws in central potential
- Path integrals for work and potential
- Simulate gravity within Solar System.

PHYSICS OF GRAVITATIONAL FIELD

Bazične formule centralnog potencijala, Keplerovi zakoni

Verificirati numerički simulacijom two-body problem

Ali, standardni RungeKutta neće biti baš dobar!

Trebati će nam symplectic solvers … u idućem poglavlju

FIELD OPERATIONS

Gravity is an excellent example of a simple **field**.

Contour plot of gravity potential as basis for gradient definition

Gradient

Divergence

Curl

Laplacian

Implementation in C++

Scalar field operations – gradient and Laplacian

```
namespace ScalarFieldOperations
{
  // ...
  template<int N>
  static VectorN<Real, N> Gradient(IScalarFunction<N>& scalarField, const VectorN<Real, N>& pos,
                                   const MetricTensorField<N>& metricTensorField) { ... }

  template<int N>
  static Real Divergence(const IVectorFunction<N>& vectorField, const VectorN<Real, N>& pos,
                         const MetricTensorField<N>& metricTensorField) { ... }

  // ...
  template<int N>
  static VectorN<Real, N> GradientCart(const IScalarFunction<N>& scalarField, const VectorN<Real, N>& pos)
  template<int N>
  static VectorN<Real, N> GradientCart(const IScalarFunction<N>& scalarField, const VectorN<Real, N>& pos,
                                       int der_order) { ... }

  static Vec3Sph GradientSpher(const IScalarFunction<3>& scalarField, const Vec3Sph& pos) { ... }
  static Vec3Sph GradientSpher(const IScalarFunction<3>& scalarField, const Vec3Sph& pos,
                               int der_order) { ... }

  static Vec3Cyl GradientCyl(const IScalarFunction<3>& scalarField, const Vec3Cyl& pos) { ... }
  static Vec3Cyl GradientCyl(const IScalarFunction<3>& scalarField, const Vec3Cyl& pos,
                             int der_order) { ... }

  // ...
  template<int N>
  static Real LaplacianCart(const IScalarFunction<N>& scalarField, const VectorN<Real, N>& pos) { ... }
  static Real LaplacianSpher(const IScalarFunction<3>& scalarField, const Vec3Sph& pos) { ... }
  static Real LaplacianCyl(const IScalarFunction<3>& scalarField, const Vec3Cyl& pos) { ... }
};
```

Calculation of a gradient in spherical coordinates

```
static Vec3Sph GradientSpher(const IScalarFunction<3>& scalarField, const Vec3Sph& pos)
{
  Vector3Spherical ret = Derivation::DerivePartialAll<3>(scalarField, pos, nullptr);

  ret[1] = ret[1] / pos[0];
  ret[2] = ret[2] / (pos[0] * sin(pos[1]));

  return ret;
}
```

VectorField operations – divergence & curl

```
namespace VectorFieldOperations
{
  // ...
  template<int N>
  static Real DivCart(const IVectorFunction<N>& vectorField, const VectorN<Real, N>& pos) { ...
  static Real DivSpher(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& x) { ...
  static Real DivCyl(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& x) { ... }

  // ...
  static Vec3Cart CurlCart(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& pos)
  static Vec3Sph  CurlSpher(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& pos)
  static Vec3Cyl  CurlCyl(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& pos) {
};
```

Calculating divergence in spherical coordinates.

```
static Real DivSpher(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& x)
{
  VectorN<Real, 3> vals = vectorField(x);

  VectorN<Real, 3> derivs;
  for (int i = 0; i < 3; i++)
    derivs[i] = Derivation::DeriveVecPartial<3>(vectorField, i, i, x, nullptr);

  Real div = 0.0;
  div += 1 / (x[0] * x[0]) * (2 * x[0] * vals[0] + x[0] * x[0] * derivs[0]);
  div += 1 / (x[0] * sin(x[1])) * (cos(x[1]) * vals[1] + sin(x[1]) * derivs[1]);
  div += 1 / (x[0] * sin(x[1])) * derivs[2];

  return div;
}
```

Calculating curl

```
static Vec3Cart CurlCart(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& pos)
{
  Real dzdy = Derivation::DeriveVecPartial<3>(vectorField, 2, 1, pos, nullptr);
  Real dydz = Derivation::DeriveVecPartial<3>(vectorField, 1, 2, pos, nullptr);

  Real dxdz = Derivation::DeriveVecPartial<3>(vectorField, 0, 2, pos, nullptr);
  Real dzdx = Derivation::DeriveVecPartial<3>(vectorField, 2, 0, pos, nullptr);

  Real dydx = Derivation::DeriveVecPartial<3>(vectorField, 1, 0, pos, nullptr);
  Real dxdy = Derivation::DeriveVecPartial<3>(vectorField, 0, 1, pos, nullptr);

  Vector3Cartesian curl{ dzdy - dydz, dxdz - dzdx, dydx - dxdy };

  return curl;
}
static Vec3Sph  CurlSpher(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& pos)
{
  VectorN<Real, 3> vals = vectorField(pos);

  Real dphidtheta = Derivation::DeriveVecPartial<3>(vectorField, 2, 1, pos, nullptr);
  Real dthetadphi = Derivation::DeriveVecPartial<3>(vectorField, 1, 2, pos, nullptr);

  Real drdphi = Derivation::DeriveVecPartial<3>(vectorField, 0, 2, pos, nullptr);
  Real dphidr = Derivation::DeriveVecPartial<3>(vectorField, 2, 0, pos, nullptr);

  Real dthetadr = Derivation::DeriveVecPartial<3>(vectorField, 1, 0, pos, nullptr);
  Real drdtheta = Derivation::DeriveVecPartial<3>(vectorField, 0, 1, pos, nullptr);

  Vector3Spherical ret;
  const Real& r = pos[0];
  const Real& theta = pos[1];
  const Real& phi = pos[2];

  ret[0] = 1 / (r * sin(theta)) * (cos(theta) * vals[2] + sin(theta) * dphidtheta - dthetadphi);
  ret[1] = 1 / r * (1 / sin(theta) * drdphi - vals[2] - r * dphidr);
  ret[2] = 1 / r * (vals[1] + r * dthetadr - drdtheta);

  return ret;
}
```

PATH INTEGRATION

Calculating work between two points along different curves connecting them, and verify with potential calculation

SIMULATING GRAVITY IN SOLAR SYSTEM

Setup of all planet orbits and simulation of satellite motion through it (Voyager?).

GRAVITATIONAL SLINGSHOT – HOW IT WORKS?

9. Simulating gravity properly in N-body problem

Deep dive with gravity simulation.

Tasks at hand:

- Numerically simulate N-body problem
- Symplectic integrators

NEED FOR SYMPLECTIC ODE SOLVERS

In time independent Hamiltonian system, the energy and the phase space volume are conserved and special integration methods have to be applied in order to ensure these conservation laws.

SIMULATING N-BODY GRAVITATIONAL PROBLEM IN C++

Modeling gravity mass and N-body configuration, with initial conditions.

```
class GravityMass
{
public:
  Real  _mass;
  Vector3Cartesian _position;
  Vector3Cartesian _velocity;

  GravityMass(const Real& mass, const Vector3Cartesian& position)
    : _mass(mass), _position(position) {}
  GravityMass(const Real& mass, const Vector3Cartesian& position, const Vector3Cartesian& velocity)
    : _mass(mass), _position(position), _velocity(velocity) {}
};

class NBodyGravityConfig
{
  std::vector<GravityMass> _masses;
public:
  int NumBodies() const { return (int)_masses.size(); }

  void AddBody(Real mass, Vector3Cartesian position, Vector3Cartesian velocity)
  {
    _masses.push_back(GravityMass(mass, position, velocity));
  }

  Real             Mass(int i)     const { return _masses[i]._mass; }
  Vector3Cartesian Position(int i) const { return _masses[i]._position; }
  Vector3Cartesian Velocity(int i) const { return _masses[i]._velocity; }

  void SetPosition(int i, Vector3Cartesian pos) { _masses[i]._position = pos; }
  void SetVelocity(int i, Vector3Cartesian vel) { _masses[i]._velocity = vel; }
};
```

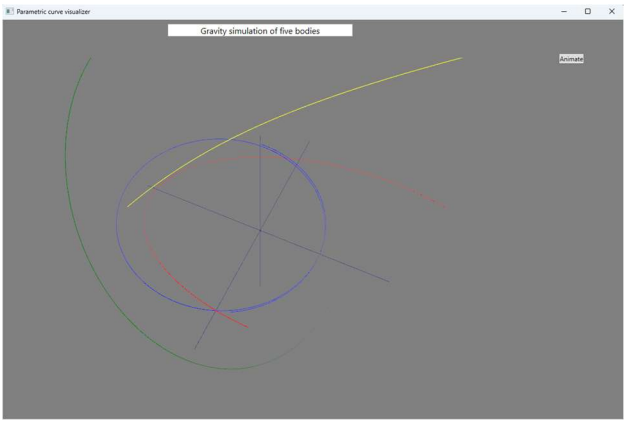
Simple Euler solver

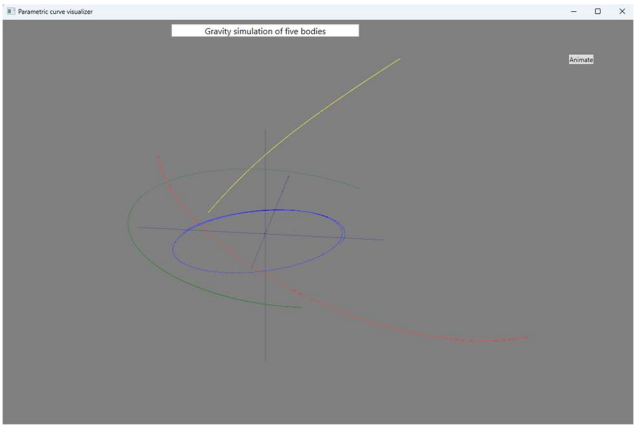
```
class NBodyMotionSolverEuler
{
protected:
  NBodyGravityConfig _config;

public:
  NBodyMotionSolverEuler(NBodyGravityConfig inConfig) : _config(inConfig) { }

  void simulateOneStep(const Real dt)
  {
    std::vector<Vector3Cartesian> force(_config.NumBodies(), Vector3Cartesian(0.0, 0.0, 0.0));

    for (int i = 0; i < _config.NumBodies(); i++) {
      // calculate force on body i
      for (int j = 0; j < _config.NumBodies(); j++) {
        if (i != j) {
          Real gravityConstant = 30;
          Vec3Cart radialVec = _config.Position(i) - _config.Position(j);
          force[i] = force[i] - gravityConstant * _config.Mass(i) * _config.Mass(j) / POW3(radialVec.NormL2()) * radialVec;
        }
      }
    }
    // advancing velocities and positions,  using the most simple Euler method of first order
    for (int i = 0; i < _config.NumBodies(); i++) {
      _config.SetVelocity(i, _config.Velocity(i) + force[i] * dt / _config.Mass(i));
      _config.SetPosition(i, _config.Position(i) + _config.Velocity(i) * dt);
    }
  }
  std::vector<std::vector<VectorN< Real, 3 >>> simulate(const Real dt, const int steps)
  {
    std::vector<std::vector<VectorN<Real,3>>> trajectories(_config.NumBodies());

    // saving initial positions
    for (int i = 0; i < _config.NumBodies(); i++)
      trajectories[i].push_back(_config.Position(i));

    for (int i = 0; i < steps; i++) {
      simulateOneStep(dt);

      for (int i = 0; i < _config.NumBodies(); i++)
        trajectories[i].push_back(_config.Position(i));
    }
    return trajectories;
  }
};
```

Main program, with setup, simulation and visualization

```
NBodyGravityConfig config;
config.AddBody(10000, Vector3Cartesian{ 0.0,    0.0,    0.0 }, Vector3Cartesian{ 0.0,   0.0,   0.0 });
config.AddBody(20, Vector3Cartesian{ -110.0,  -50.0,   10.0 }, Vector3Cartesian{ 0.0,   50,   0.0 });
config.AddBody(10, Vector3Cartesian{ 130.0,   50.0,   70.0 }, Vector3Cartesian{ 0.0,   -50,   0 });
config.AddBody(20, Vector3Cartesian{ -20.0,  100.0, -110.0 }, Vector3Cartesian{ 50,   0.0,   0.0 });
config.AddBody(10, Vector3Cartesian{ 70.0, -110.0,   70.0 }, Vector3Cartesian{ -50,   50,   50.0 });

NBodyMotionSolverEuler solver(config);

const Real dt    = 0.01;
const int  steps = 2000;

auto res = solver.simulate(dt, steps);

for (int i = 0; i < config.NumBodies(); i++)
{
  Serializer::SaveAsParamCurve<3>(res[i], "PARAMETRIC_CURVE_CARTESIAN_3D", "Body" + std::to_string(i+1),
                                  0.0, dt * steps, steps + 1,
                                  MML_PATH_ResultFiles + "body" + std::to_string(i) + ".txt");
}

Visualizer::VisualizeMultiParamCurve3D({"body0.txt", "body1.txt", "body2.txt", "body3.txt", "body4.txt" });
```

Resulting trajectories





Defining ODE system for N-body problem

```
class NBodySystemODE : public IODESystem
{
  NBodyGravityConfig _config;
  const Real G = 6.67430e-11;

public:
  NBodySystemODE(NBodyGravityConfig inConfig) : _config(inConfig) { }

  int getDim() const { return 3 * _config.NumBodies(); }
  void derivs(const Real t, const Vector<Real>& x, Vector<Real>& dxdt) const
  {
    // filling in dxdt vector with N * 6 derivations of our variables (x, y, z, vx, vy, vz)
    for (int i = 0; i < _config.NumBodies(); i++)
    {
      // calculating force on body i
      Vector3Cartesian force(0, 0, 0);
      for (int j = 0; j < _config.NumBodies(); j++)
      {
        if (i != j)      // checking for self-force
        {
          Vector3Cartesian vec_dist(dxdt[6 * j] - dxdt[6 * i],
                                    dxdt[6 * j + 2] - dxdt[6 * i + 2],
                                    dxdt[6 * j + 4] - dxdt[6 * i + 4]);

          force = force - G * _config.Mass(i) * _config.Mass(j) / POW3(vec_dist.NormL2()) * vec_dist;
        }
      }

      // x coord
      dxdt[6 * i]     = x[6 * i + 1];
      dxdt[6 * i + 1] = 1 / _config.Mass(i) * force.X();
      // y coord
      dxdt[6 * i + 2] = x[6 * i + 3];
      dxdt[6 * i + 3] = 1 / _config.Mass(i) * force.Y();
      // z coord
      dxdt[6 * i + 4] = x[6 * i + 5];
      dxdt[6 * i + 5] = 1 / _config.Mass(i) * force.Z();
    }
  }
};
```

Main program

TODO

Using symplectic integrators!

Runge-Kutta-Nystroem solvers

Visualizing fields

10. Coordinate transformations

Contravariant and covariant vectors … and all that jazz.

Tasks at hand:

- Modelling general coordinate transformations in C++
- Implement various transformations
- Oblique systems and difference between contravariant and covariant vectors

TRANSFORMATION OF COORDINATES

General transformation of coordinates:

It is assumed we also have an inverse transformation

Rotations in 2D

Orthogonal transformations

Curvilinear

Spherical

Mathematical and physical convention in ordering coordinates.

Cylindrical

Oblique Cartesian coordinate system

Dual basis

TRANSFORMING VECTORS

Covariant and contravariant vectors

Vector3Spherical

Vector3Cylindrical

IMPLEMENTATIONS IN C++

We define two basic interfaces.

Why inheriting from IVectorFunction? Because every coordinate transformation is actually defined through vector function – function that takes vector of original coordinates, and returns vector of transformed coordinates.

```
template<typename VectorFrom, typename VectorTo, int N>
class ICoordTransf : public IVectorFunction<N>
{
public:
  virtual       VectorTo            transf(const VectorFrom& in) const = 0;
  virtual const IScalarFunction<N>& coordTransfFunc(int i) const = 0;

  virtual ~ICoordTransf() {}
};
```

For transformations that have an inverse, we specialize original interface, requiring

```
template<typename VectorFrom, typename VectorTo, int N>
class ICoordTransfWithInverse : public virtual ICoordTransf<VectorFrom, VectorTo, N>
{
public:
  virtual       VectorFrom          transfInverse(const VectorTo& in) const = 0;
  virtual const IScalarFunction<N>& inverseCoordTransfFunc(int i) const = 0;

  virtual ~ICoordTransfWithInverse() {}
};
```

We need explicit form of each transformation function, so we can do mathematical operations on those functions.

Then we create two abstract classes,

```
template<typename VectorFrom, typename VectorTo, int N>
class CoordTransf : public virtual ICoordTransf<VectorFrom, VectorTo, N>
{
public:
  // inherited from IVectorFunction
  VectorN<Real, N> operator()(const VectorN<Real, N>& x) const
  {
    VectorN<Real, N> ret;
    for (int i = 0; i < N; i++)
      ret[i] = this->coordTransfFunc(i)(x);
    return ret;
  }

  virtual VectorTo   getBasisVec(int ind, const VectorFrom& pos) { ... }
  virtual VectorFrom getInverseBasisVec(int ind, const VectorFrom& pos) { ... }

  MatrixNM<Real, N, N> jacobian(const VectorN<Real, N>& pos) { ... }

  VectorTo   transfVecContravariant(const VectorFrom& vec, const VectorFrom& pos) { ... }
  VectorFrom transfInverseVecCovariant(const VectorTo& vec, const VectorFrom& pos) { ... }
};
```

```
template<typename VectorFrom, typename VectorTo, int N>
class CoordTransfWithInverse : public virtual CoordTransf<VectorFrom, VectorTo, N>,
                               public virtual ICoordTransfWithInverse<VectorFrom, VectorTo, N>
{
public:
  virtual VectorFrom getContravarBasisVec(int ind, const VectorTo& pos) { ... }
  virtual VectorTo   getInverseContravarBasisVec(int ind, const VectorTo& pos) { ... }

  VectorTo   transfVecCovariant(const VectorFrom& vec, const VectorTo& pos) { ... }
  VectorFrom transfInverseVecContravariant(const VectorTo& vec, const VectorTo& pos) { ... }

  Tensor2<N> transfTensor2(const Tensor2<N>& tensor, const VectorFrom& pos) { ... }
  Tensor3<N> transfTensor3(const Tensor3<N>& tensor, const VectorFrom& pos) { ... }
  Tensor4<N> transfTensor4(const Tensor4<N>& tensor, const VectorFrom& pos) { ... }
  Tensor5<N> transfTensor5(const Tensor5<N>& tensor, const VectorFrom& pos) { ... }
};
```

Finally, example implementation for concrete spherical to cartesian transformation.

```
class CoordTransfSphericalToCartesian : public CoordTransfWithInverse<Vector3Spherical, Vector3Cartesian, 3>
{
private:
  // q[0] = r     - radial distance
  // q[1] = theta - inclination
  // q[2] = phi   - azimuthal angle
  static Real x(const VectorN<Real, 3>& q) { return q[0] * sin(q[1]) * cos(q[2]); }
  static Real y(const VectorN<Real, 3>& q) { return q[0] * sin(q[1]) * sin(q[2]); }
  static Real z(const VectorN<Real, 3>& q) { return q[0] * cos(q[1]); }

  // ...
  static Real r(const VectorN<Real, 3>& q)     { return sqrt(q[0]*q[0] + q[1]*q[1] + q[2]*q[2]); }
  static Real theta(const VectorN<Real, 3>& q) { return acos(q[2] / sqrt(q[0]*q[0] + q[1]*q[1] + q[2]*q[2])); }
  static Real phi(const VectorN<Real, 3>& q)   { return atan2(q[1], q[0]); }

  inline static ScalarFunction<3> _func[3] = { ScalarFunction<3>{x},
                                               ScalarFunction<3>{y},
                                               ScalarFunction<3>{z}
  };

  inline static ScalarFunction<3> _funcInverse[3] = { ScalarFunction<3>{r},
                                                      ScalarFunction<3>{theta},
                                                      ScalarFunction<3>{phi}
  };
public:
  Vector3Cartesian     transf(const Vector3Spherical& q)        const { return Vector3Cartesian{ x(q), y(q), z(q) }; }
  Vector3Spherical     transfInverse(const Vector3Cartesian& q) const { return Vector3Spherical{ r(q), theta(q), phi(q) }; }

  const IScalarFunction<3>& coordTransfFunc(int i)        const { return _func[i]; }
  const IScalarFunction<3>& inverseCoordTransfFunc(int i) const { return _funcInverse[i]; }
```

Example of a RotationXAxis transformation, that unlike spherical transformation, depends on a parameter (angle of rotation).

```
class CoordTransfCart3DRotationXAxis :
  public CoordTransfWithInverse<Vector3Cartesian, Vector3Cartesian, 3>
{
private:
  Real    _angle;
  MatrixNM<Real, 3, 3>  _transf;
  MatrixNM<Real, 3, 3>  _inverse;

  const ScalarFunctionFromStdFunc<3> _f1, _f2, _f3;
  const ScalarFunctionFromStdFunc<3> _fInverse1, _fInverse2, _fInverse3;
```

Fun begins with the constructor:

```
CoordTransfCart3DRotationXAxis(Real inAngle) : _angle(inAngle),
  _f1(std::function<Real(const VectorN<Real, 3>&)>
        { std::bind(&CoordTransfCart3DRotationXAxis::func1, this, std::placeholders::_1) }
    ),
  _f2(std::function<Real(const VectorN<Real, 3>&)>
        { std::bind(&CoordTransfCart3DRotationXAxis::func2, this, std::placeholders::_1) }
    ),
  _f3(std::function<Real(const VectorN<Real, 3>&)>
        { std::bind(&CoordTransfCart3DRotationXAxis::func3, this, std::placeholders::_1) }
    ),
  _fInverse1(std::function<Real(const VectorN<Real, 3>&)>
        { std::bind(&CoordTransfCart3DRotationXAxis::funcInverse1, this, std::placeholders::_1) }
    ),
  _fInverse2(std::function<Real(const VectorN<Real, 3>&)>
        { std::bind(&CoordTransfCart3DRotationXAxis::funcInverse2, this, std::placeholders::_1) }
    ),
  _fInverse3(std::function<Real(const VectorN<Real, 3>&)>
        { std::bind(&CoordTransfCart3DRotationXAxis::funcInverse3, this, std::placeholders::_1) }
    )
{
  _transf[0][0] = 1.0;
  _transf[1][1] = cos(_angle);
  _transf[1][2] = -sin(_angle);
  _transf[2][1] = sin(_angle);
  _transf[2][2] = cos(_angle);

  _inverse[0][0] = 1.0;
  _inverse[1][1] = cos(_angle);
  _inverse[1][2] = sin(_angle);
  _inverse[2][1] = -sin(_angle);
  _inverse[2][2] = cos(_angle);
}
```

11. All is not well, if you are in a non-inertial frame!

Wouldn't you know, there are some "fictious" forces in classical mechanics?

Tasks at hand:

- Investigating centrifugal force on a simple 2D carousel

PHYSICS

SIMPLE CAROUSEL AND CENTRIFUGAL FORCE

Carousel with radius of 20 meters, throwing darts

3 pikada, ako baci točno u centar (u njeovom coord sustavu), gdje će strelica završiti

INTRODUCING REFERENTIALFRAME

Carousel on carousel ☺

Shooting method za ODE's?

12. Projectile launch

Launching projectiles … what could be more fun.

Tasks at hand:

- A projectile is fired from the center of Ban Jelačić Square in Zagreb (45°48'47.4"N 15°58'38.3"E) at an angle of 45 degrees, in eastward direction. Ignoring air resistance, what is position of the projectile after one hour, for following values of the initial velocity (in m/s): 200, 1.000, 10.000, 15.000?
- What if we include air resistance?
- How can we simulate rocket motors (ie. the "projectile" doesn't get all its speed at the start)

ARTILLERY GRENADE – 200 M/S

Coriolis force enters the scene!

BALLISTIC ROCKET – 1000 M/S

LOW ORBIT SATELLITE – 10000 M/S

HITTING THE MOON AND BEYOND – 15000 M/S

We will revisit this problem for special relativity velocities

13. Rigid body

Point-like mechanical particles are great … but there is more to mechanics than that.

Task at hand:

- Simple body pushed with some force in space without gravity, how it moves and rotates?
- Spinning top without gravity
- Tensors
- Eigenvalues

introducing tensors and their transformations

PHYSICS OF RIGID BODY

Moment of inertia

Eigenvalues

Euler equations

TENSORS

Basic interface for rank 2 tensor (similar defined up to rank 5).

```
template<int N>
class ITensor2
{
public:
  virtual int   NumContravar() const = 0;
  virtual int   NumCovar() const = 0;

  virtual Real  operator()(int i, int j) const = 0;
  virtual Real& operator()(int i, int j) = 0;
};
```

Concrete implementation

```
template <int N>
class Tensor2 : public ITensor2<N>
{
  MatrixNM<Real, N, N> _coeff;
public:
  int _numContravar = 0;
  int _numCovar = 0;
  bool _isContravar[2];

  Tensor2(int nCovar, int nContraVar) { ... }
  Tensor2(int nCovar, int nContraVar, std::initializer_list<Real> values) { ... }

  int  NumContravar() const { return _numContravar; }
  int  NumCovar()     const { return _numCovar; }

  bool IsContravar(int i) const { return _isContravar[i]; }
  bool IsCovar(int i) const     { return !_isContravar[i]; }

  Real  operator()(int i, int j) const override { return _coeff[i][j]; }
  Real& operator()(int i, int j) override       { return _coeff[i][j]; }

  Tensor2 operator+(const Tensor2& other) const { ... }
  Tensor2 operator-(const Tensor2& other) const { ... }
  Tensor2 operator*=(Real scalar) const { ... }
  Tensor2 operator/(Real scalar) const { ... }

  friend Tensor2 operator*(Real scalar, const Tensor2& b) { ... }

  Real Contract() const { ... }
  Real operator()(const VectorN<Real, N>& v1, const VectorN<Real, N>& v2) const { ... }

  void   Print(std::ostream& stream, int width, int precision) const { ... }
  friend std::ostream& operator<<(std::ostream& stream, const Tensor2& a) { ... }
};
```

TENSOR TRANSFORMATIONS

Transformation of rank 2 tensor.

Member of CoordTransfWithInverse abstract class, and available for every inherited coordinate transformation class.

```
Tensor2<N> transfTensor2(const Tensor2<N>& tensor, const VectorFrom& pos)
{
  Tensor2<N> ret(tensor.NumContravar(), tensor.NumCovar());

  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
    {
      ret(i, j) = 0;
      for (int k = 0; k < N; k++)
        for (int l = 0; l < N; l++)
        {
          double coef1, coef2;
          if (tensor._isContravar[0])
            coef1 = Derivation::NDer1Partial(this->coordTransfFunc(i), k, pos);
          else
            coef1 = Derivation::NDer1Partial(this->inverseCoordTransfFunc(k), i, pos);

          if (tensor._isContravar[1])
            coef2 = Derivation::NDer1Partial(this->coordTransfFunc(j), l, pos);
          else
            coef2 = Derivation::NDer1Partial(this->inverseCoordTransfFunc(l), j, pos);

          ret(i, j) += coef1 * coef2 * tensor(k, l);
        }
    }

  return ret;
}
```

CALCULATING MOMENT OF INERTIA

Calculating moment of inertia for a set of discrete masses

Necessary models:

```
struct DiscreteMass {
    Vector3Cartesian _position;
    double _mass;

    DiscreteMass(const Vector3Cartesian &position, const double& mass)
        : _position(position), _mass(mass)  { }
};
struct DiscreteMassesConfig {
    std::vector<DiscreteMass> _masses;

    DiscreteMassesConfig(const std::vector<DiscreteMass>& masses)
        : _masses(masses)  { }
};
```

Calculator

```
class DiscreteMassMomentOfInertiaTensorCalculator
{
    DiscreteMassesConfig _massesConfig;
public:
    DiscreteMassMomentOfInertiaTensorCalculator(const DiscreteMassesConfig& massesConfig)
        : _massesConfig(massesConfig) { }

    Tensor2<3> calculate()
    {
        Tensor2<3> tensor(2,0);  // can be (0,2) or (1,1) as well (it is a Cartesian tensor)
        for (const auto& mass : _massesConfig._masses)
        {
            Vector3Cartesian pos = mass._position;
            tensor(0,0) += mass._mass * (pos.Y() * pos.Y() + pos.Z() * pos.Z());
            tensor(1,1) += mass._mass * (pos.X() * pos.X() + pos.Z() * pos.Z());
            tensor(2,2) += mass._mass * (pos.X() * pos.X() + pos.Y() * pos.Y());

            tensor(0,1) -= mass._mass * pos.X() * pos.Y();
            tensor(0,2) -= mass._mass * pos.X() * pos.Z();
            tensor(1,2) -= mass._mass * pos.Y() * pos.Z();

            tensor(1,0) = tensor(0,1);
            tensor(2,0) = tensor(0,2);
            tensor(2,1) = tensor(1,2);
        }
        return tensor;
    }
};
```

Calculating moment of inertia for continuous mass

CALCULATING EIGENVALUES

SIMULATING RIGID BODY

14. Rotations and quaternions

Orientation, orientation, orientation … and transformation.

Tasks at hand:

- Rotations
- Quaternions

REPRESENTING ROTATIONS

QUATERNIONS

15. Motion in spacetime – Lorentz transformations

When your speed approaches speed of light, strange things tend to happen.

Tasks at hand:

- Basics of special relativity and Lorentz transformations
- Introduce Vector4Lorentz and LorentzTransformation classes

PHYSICS – BASICS OF SPECIAL RELATIVITY

VECTOR4LORENTZ

Adding time as coordinate for specifying vectors in four-dimensional spacetime.

Per almost universal custom, time coordinate is first one, with convenient index 0.

```
class Vector4Lorentz : public VectorN<Real, 4>
{
public:
  Real  T() const { return _val[0]; }
  Real& T()       { return _val[0]; }
  Real  X() const { return _val[1]; }
  Real& X()       { return _val[1]; }
  Real  Y() const { return _val[2]; }
  Real& Y()       { return _val[2]; }
  Real  Z() const { return _val[3]; }
  Real& Z()       { return _val[3]; }

  Vector4Lorentz() : VectorN<Real, 4>{ 0.0, 0.0, 0.0, 0.0 } {}
  Vector4Lorentz(std::initializer_list<Real> list) : VectorN<Real, 4>(list) { }
};
```

LORENTZTRANSFORMATION

TODO – general Lorentz transformation, for a boost in any direction

INTRODUCING METRIC TENSOR

Interface for rank two tensor field in N dimensions.

```
template<int N>
class ITensorField2 : public IFunction<Tensor2<N>, const VectorN<Real, N>& >
{
  int _numContravar;
  int _numCovar;
public:
  ITensorField2(int numContra, int numCo) : _numContravar(numContra), _numCovar(numCo) { }

  int getNumContravar() const { return _numContravar; }
  int getNumCovar()     const { return _numCovar; }

  // concrete implementations need to provide this
  virtual Real Component(int i, int j, const VectorN<Real, N>& pos) const = 0;

  virtual ~ITensorField2() {}
};
```

Abstract class representing general metric tensor field, defined throughout the whole space.

```
template<int N>
class MetricTensorField : public ITensorField2<N>
{
public:
  MetricTensorField() : ITensorField2<N>(2, 0) { }
  MetricTensorField(int numContra, int numCo) : ITensorField2<N>(numContra, numCo) { }

  // implementing operator() requried by IFunction interface
  Tensor2<N>   operator()(const VectorN<Real, N>& pos) const
  {
    Tensor2<N> ret(this->getNumContravar(), this->getNumCovar());

    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        ret(i, j) = this->Component(i, j, pos);

    return ret;
  }

  Real GetChristoffelSymbolFirstKind(int i, int j, int k, const VectorN<Real, N>& pos) const
  Real GetChristoffelSymbolSecondKind(int i, int j, int k, const VectorN<Real, N>& pos) const

  VectorN<Real, N> CovariantDerivativeContravar(const IVectorFunction<N>& func, int j,
                                                const VectorN<Real, N>& pos) const { ... }
  Real CovariantDerivativeContravarComp(const IVectorFunction<N>& func, int i, int j,
                                        const VectorN<Real, N>& pos) const { ... }

  VectorN<Real, N> CovariantDerivativeCovar(const IVectorFunction<N>& func, int j,
                                            const VectorN<Real, N>& pos) const { ... }
  Real CovariantDerivativeCovarComp(const IVectorFunction<N>& func, int i, int j,
                                    const VectorN<Real, N>& pos) const { ... }
};
```

Simplest example is metric tensor for Cartesian space in three dimensions.

```
class MetricTensorCartesian3D : public MetricTensorField<3>
{
public:
  MetricTensorCartesian3D() : MetricTensorField<3>(2, 0) { }

  Real Component(int i, int j, const VectorN<Real, 3>& pos) const
  {
    if (i == j)
      return 1.0;
    else
      return 0.0;
  }
};
```

Metric tensor for spherical coordinates comes in two variants.

Covariant

```
class MetricTensorSpherical : public MetricTensorField<3>
{
public:
  MetricTensorSpherical() : MetricTensorField<3>(0, 2) { }

  virtual  Real Component(int i, int j, const VectorN<Real, 3>& pos) const override
  {
    if (i == 0 && j == 0)
      return 1.0;
    else if (i == 1 && j == 1)
      return POW2(pos[0]);
    else if (i == 2 && j == 2)
      return pos[0] * pos[0] * sin(pos[1]) * sin(pos[1]);
    else
      return 0.0;
  }
};
```

And contravariant (which is just the inverse of covariant version).

```
class MetricTensorSphericalContravar : public MetricTensorField<3>
{
public:
  MetricTensorSphericalContravar() : MetricTensorField<3>(2, 0) { }

  virtual Real Component(int i, int j, const VectorN<Real, 3>& pos) const
  {
    if (i == 0 && j == 0)
      return 1.0;
    else if (i == 1 && j == 1)
      return 1 / (pos[0] * pos[0]);
    else if (i == 2 && j == 2)
      return 1 / (pos[0] * pos[0] * sin(pos[1]) * sin(pos[1]));
    else
      return 0.0;
  }
};
```

Finally, we come to metric tensor for spacetime, which we will call MetricTensorMinkowski:

```
class MetricTensorMinkowski : public MetricTensorField<4>
{
public:
  MetricTensorMinkowski() : MetricTensorField<4>(2, 0) {}

  virtual Real Component(int i, int j, const VectorN<Real, 4>& pos) const override
  {
    if (i == 0 && j == 0)
      return -1.0;
    else if (i == 1 && j == 1)
      return 1.0;
    else if (i == 2 && j == 2)
      return 1.0;
    else if (i == 3 && j == 3)
      return 1.0;
    else
      return 0.0;
  }
};
```

SOLVED EXAMPLES

Projectile launch with relativistic speed

Continuing our example from Chapter 12, what is position if we launch it with speeds: 0.1c, 0.3c, 0.5c, 0.8c, 0.9c, 0.95c?

Earth gravity and Coriolis force are not relevant here, and the only important

Passenger on train dropping ball

Moving on a train with 0.8c, passenger drops a ball to the floor. At what time it hits the floor for passenger, and stationary observer

Spherical ball passing by observer with relativistic speed

Spherical ball, 1 meter in diameter, is travelling with speed 0.9c along a line that at its closest is 100 meters from observer.

Observer has an extremely fast camera with a tracker that ensures camera is pointed exactly at the centre of the sphere at all times.

16. Special relativity – resolving twin-paradox

Where we investigate the most famous "paradox" of them all.

Task at hand:

- Simulate numerically twin paradox
- Simple case with linear trajectory from A to B
- Realistic case with acceleration and a real trajectory to destination

PHYSICS OF PROPER TIME

NUMERICAL SIMULATION

17. Static electric fields

Starting our investigations in electromagnetism.

Task at hand:

- Calculate electric fields for different charge configurations.
- Verify Gauss and Stokes law, both in integral and differential form.

PHYSICS – COULOMB LAW

Jednostavno numeričko izračunavanje potencijala za analitički poznata rješenja I usporedba

SIMULATING DISTRIBUTION OF CHARGE ON A SOLID BODY

What is distribution of charge

Sfera

Cilindar

Kvadar

Something with “pointy”

ELECTRIC FIELD OF A ROD WITH FINITE LENGTH

In previous section we calculated distribution of charge

Which means we can now calculate electric field throughout space

But actually, we want field lines visualized

Projection in plane

POTENTIAL AND WORK IN ELECTRIC FIELD

CONDUCERS AND DIELECTRICS

18. Static magnetic fields

Looking at static magnetic fields.

Task at hand:

- Calculate magnetic fields for different current configurations.

PHYSICS - BIOT-SAVART LAW

Magnetic field of a simple loop with passing current

19. Dynamic EM fields

What happens when charges and currents are not static?

Task at hand:

- Investigate dynamic electromagnetic fields.
- Introduce electro-magnetic tensor
- EM field of a moving charge

PHYSICS – MAXWELL'S EQUATIONS

INTRODUCING EM TENSOR

```
Tensor2<4> GetEMTensorContravariant(Vector3Cartesian E_field, Vector3Cartesian B_field)
{
  double c = 1.0;

  Tensor2<4> EM_tensor(2, 0);

  EM_tensor(0, 0) = 0.0;
  EM_tensor(0, 1) = -E_field.X() / c;
  EM_tensor(0, 2) = -E_field.Y() / c;
  EM_tensor(0, 3) = -E_field.Z() / c;

  EM_tensor(1, 0) = E_field.X() / c;
  EM_tensor(1, 1) = 0.0;
  EM_tensor(1, 2) = -B_field.Z();
  EM_tensor(1, 3) = B_field.Y();

  EM_tensor(2, 0) = E_field.Y() / c;
  EM_tensor(2, 1) = B_field.Z();
  EM_tensor(2, 2) = 0.0;
  EM_tensor(2, 3) = -B_field.X();

  EM_tensor(3, 0) = E_field.Z() / c;
  EM_tensor(3, 1) = -B_field.Y();
  EM_tensor(3, 2) = B_field.X();
  EM_tensor(3, 3) = 0.0;

  return EM_tensor;
}
```

LIENARD-WIECHERT POTENTIAL OF MOVING CHARGE

Basic calculation

```
// Given t and r in lab frame of the observer, calculate Lienard-Wiechert potentials
// for charge moving along x-axis with given velocity
Real calcLienardWiechertScalarPotential(Vector3Cartesian r_at_point, Real t, Vector3Cartesian rs_charge_pos,
                                        Real q, Real charge_velocity)
{
  double c = 3e8;
  Vec3Cart charge_v(charge_velocity, 0, 0);

  // calculate retarded time
  Real r = (r_at_point - rs_charge_pos).NormL2();
  Real tr = t - (r_at_point - rs_charge_pos).NormL2() / c;

  // calculate retarded position
  Vec3Cart r_ret_charge_pos = r_at_point - charge_v * (t - tr);

  Real beta = charge_v.NormL2() / c;
  Vec3Cart ns = (r_at_point - r_ret_charge_pos) / (r_at_point - r_ret_charge_pos).NormL2();

  Real scalar_potential = q / ((r_at_point - r_ret_charge_pos).NormL2() * (1 - beta * ScalarProduct(ns, charge_v) / c));

  return scalar_potential;
}
```

TODO;

Parametrizacija moving naboja, što mu je t?

Imamo grid postavljenih mjeritelja el. Polja, koje s brzinom c vraćaju signale nazad do observera

Ispaljuje se naboj brzinom 0.8 c u t = 0, za observera

Simulacija takva da je u prvih 10 dT, kod observera polja nema, a na 10d dT dođe prvi signal od najbližžeg mjeritelja točki ispaljivanja

SIMPLE ANTENNA?

Varying current through antenna and resulting electric field

20. Differential geometry of curves and surfaces

Diving into some differential geometry with curves and surfaces.

Task at hand:

- Implement numerical calculations of curves and surfaces differential geometry properties.
- Frenet-Serret formulas, moving trihedron
- Connection with velocity and acceleration

CURVES

Mathematics of curves

Most of the formulas are for “arc-length parametrized” curves … and those are mostly NOT what occurs in practice

TODO – give both set of formulas with intro

Implementation in C++

Starting from interface IParametricCurve

```
// abstract class, providing basic curve formulas in 2D
class ICurveCartesian2D : public IParametricCurve<2>
{
public:
  Vec2Cart getTangent(Real t)
  {
    return Derivation::DeriveCurve<2>(*this, t, nullptr);
  }
  Vec2Cart getTangentUnit(Real t)
  {
    return getTangent(t).GetAsUnitVector();
  }
  Vec2Cart getNormal(Real t)
  {
    return Derivation::DeriveCurveSec<2>(*this, t, nullptr);
  }
  Vec2Cart getNormalUnit(Real t)
  {
    return getNormal(t).GetAsUnitVector();
  }
};
```

Set of defined 2D (planar) curves

```
class Circle2DCurve : public ICurveCartesian2D
{
  Real _radius;
  Pnt2Cart _center;
public:
  Circle2DCurve() : _radius(1), _center(0, 0) {}
  Circle2DCurve(Real radius) : _radius(radius), _center(0,0) {}
  Circle2DCurve(Real radius, const Pnt2Cart& center) : _radius(radius), _center(center) {}

  Real getMinT() const { return 0.0; }
  Real getMaxT() const { return 2 * Constants::PI; }

  VectorN<Real, 2> operator()(Real t) const {
    return MML::VectorN<Real, 2>{_center.X()+_radius* cos(t), _center.Y()+_radius* sin(t)};
  }
};

class LogSpiralCurve { ... };

class LemniscateCurve { ... };

class DeltoidCurve { ... };

class AstroidCurve { ... };

class EpitrochoidCurve { ... };

class ArchimedeanSpiralCurve { ... };
```

Going to 3D, we again have an abstract class

```
// abstract class, providing basic curve formulas in 3D
class ICurveCartesian3D : public IParametricCurve<3>
{
public:
  Vec3Cart getTangent(Real t) const
  {
    return Derivation::DeriveCurve<3>(*this, t, nullptr);
  }
  Vec3Cart getTangentUnit(Real t) const
  {
    return getTangent(t).GetAsUnitVector();
  }
  Vec3Cart getNormal(Real t) const
  {
    return Derivation::DeriveCurveSec<3>(*this, t, nullptr);
  }
  Vec3Cart getNormalUnit(Real t) const
  {
    return getNormal(t).GetAsUnitVector();
  }
  Vec3Cart getBinormal(Real t) const
  {
    Vec3Cart tangent = getTangentUnit(t);
    Vec3Cart normal  = getNormalUnit(t);

    return VectorProduct(tangent, normal);
  }

  Vec3Cart getCurvatureVector(Real t) const { ... }
  Real getCurvature(Real t) const { ... }
  Real getTorsion( Real t) const { ... }
};
```

And a set of predefined 3D curves

```
class LineCurve { ... };

class Circle3DXY { ... };

class Circle3DXZ { ... };

class Circle3DYZ { ... };

class HelixCurve : public ICurveCartesian3D
{
  Real _radius, _b;
public:
  HelixCurve() : _radius(1.0), _b(1.0) {}
  HelixCurve(Real radius, Real b) : _radius(radius), _b(b) {}

  Real getMinT() const { return Constants::NegativeInf; }
  Real getMaxT() const { return Constants::PositiveInf; }

  VectorN<Real, 3> operator()(Real t) const {
    return MML::VectorN<Real, 3>{_radius * cos(t), _radius * sin(t), _b * t};
  }

  Real getCurvature(Real t) const { return _radius / (POW2(_radius) + POW2(_b)); }
  Real getTorsion(Real t) const { return _b / (POW2(_radius) + POW2(_b)); }
};

class TwistedCubicCurve { ... };

class ToroidalSpiralCurve { ... };
```

Also, a concrete class, useful if you have a function pointer, or curve is simple enough to be defined with lambda.

```
// concrete class, that can be initialized with function pointers or lambdas
class CurveCartesian3D : public ICurveCartesian3D
{
  Real _minT;
  Real _maxT;
  VectorN<Real, 3>(*_func)(Real);
public:
  CurveCartesian3D(VectorN<Real, 3>(*inFunc)(Real)) { ... }
  CurveCartesian3D(Real minT, Real maxT, VectorN<Real, 3>(*inFunc)(Real)) { ... }

  Real getMinT() const { return _minT; }
  Real getMaxT() const { return _maxT; }

  virtual VectorN<Real, 3> operator()(Real x) const { return _func(x); }
};
```

SURFACES

LOOKING TO MANIFOLDS

Sphere as a manifold

Can we calculate some geodesics?

21. General relativity

Going to the final frontier.

Task at hand:

- Schwarzschild geometry of black hole. How does crossing the event horizon looks from perspective of observer and traveler.
- How it is different for Kerr metric

EINSTEIN'S EQUATION

STATIC BLACK HOLE - SCHWARZSCHILD'S METRIC

ROTATING BLACK HOLE - KERR METRIC