Exploring Physics with C++

Zvonimir Vanjak

EXPLORING PHYSICS WITH C++

Contents overview

Only first two levels of chapters

Contents in detail

Preface

Why?

- 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++

Personal side

- 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

Goals

Using cross-platform C++, with cross-platform visualization tools (as much as possible)

Investigate

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

Hopefully, this part will be much, much expanded in the final version

Zahvale contributorima

1. Essential mathematical objects

Where we start from the beginning.

PRELIMINARIES

In this chapter we handle the essentials.

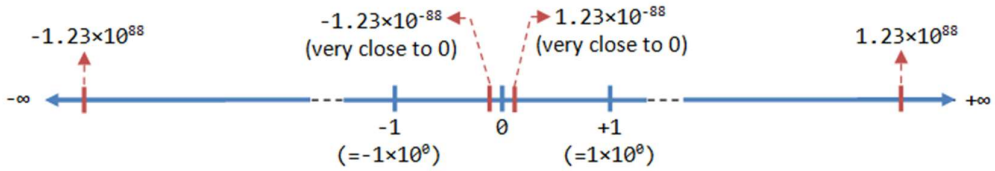
Doing numerics on computer – basics, challenges, pitfalls and traps

Representing numbers

Integers are represented precisely, but hell awaits anyone who steps out of the range of values.

Precision of real representation - about IEEE754, mantissa, exponents, and all that

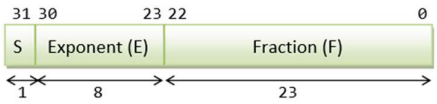
TODO – nice picture of graphical representation

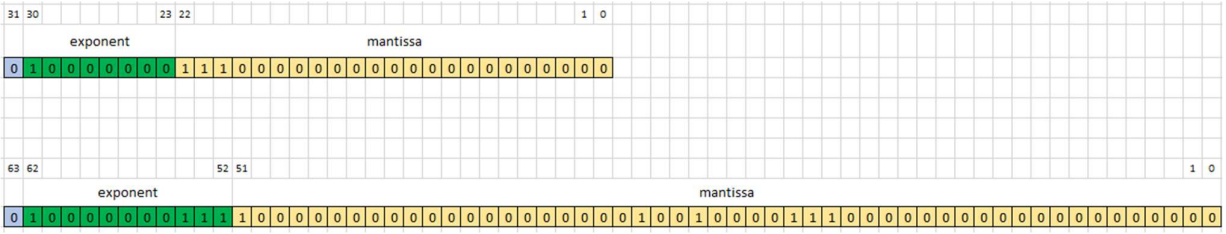


float, double, long double types in standard

representation of special values, in a table like this:

	S	*E*	*F*	Value
`float`	any	1–254	any	$(-1)^S \times 2^{E-127} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-126} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	255	0	$+\infty$
	1	255	0	$-\infty$
	any	255	nonzero	NaN
`double`	any	1–2046	any	$(-1)^S \times 2^{E-1023} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-1022} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	2047	0	$+\infty$
	1	2047	0	$-\infty$
	any	2047	nonzero	NaN
*unnormalized values				





Representing PI

Epsilon for float, double, long double (check on different systems – Win, Mac, Linux)

Compare real representation with resolution needed for measuring lengths of different scale in Solar System.

Measure everything in units of A.U. (astronomical units), where 1.0 = 150 million kilometers or 1.0 = 1.5e11 meters.

For example, Earth radius of 6.400 km in this units is equal to 4,27e-5, or 0,0000427 A.U.!

If we want to simulate something that on Earth is on the scale of 1 meter, well, 1 m = 6,67e-12, or, to give a visual representation 0,00000000000667 A.U.

For this kind of simulation, float is obviously out of the question, but even if we use double precision, we get a resolution of approximately 1 cm

Let's say we want to calculate motion of thrown ball on Pluto, but with a twist, we want to include ALL gravitational influences in Solar system!

Distance Sun – Pluto is 5,9 billion kilometers, or 5.9e12 meters

Using scale where Pluto's center is exactly at 1.00000000, and taking into account Plut's radius of 1188 km, position of point on Pluto's surface (exactly opposite to Sun) is given by 1.000000000201355.

Distance of 1 meter on Pluto's surface is in this scale equal to 1,6949152e-16, which means that even double precision is completely inadequate for modeling this distance.

__float128, existing in GCC and as Quad type in Intel C++ Compiler.

Roundoff errors

Happens because of inexact representation of real numbers.

Couple of examples of roundoff errors

Special emphasis on subtraction of similar numbers

TODO – something like this!

Suppose we wish to evaluate the function

$$f(x) = \frac{1 - cos(x)}{sin(x)}, \tag{2.7}$$

for small values of x. If we multiply the denominator and numerator with $1 + cos(x)$ we obtain the equivalent expression

$$f(x) = \frac{sin(x)}{1 + cos(x)}. \tag{2.8}$$

If we now choose $x = 0.007$ (in radians) our choice of precision results in

$$sin(0.007) \approx 0.69999 \times 10^{-2},$$

and

$$cos(0.007) \approx 0.99998.$$

The first expression for $f(x)$ results in

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2}, \tag{2.9}$$

while the second expression results in

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2}, \tag{2.10}$$

which is also the exact result. In the first expression, due to our choice of precision, we have only one relevant digit in the numerator, after the subtraction. This leads to a loss of precision and a wrong result due to a cancellation of two nearly equal numbers. If we had chosen a precision of six leading digits, both expressions yield the same answer. If we were to evaluate $x \sim \pi$, then the second expression for $f(x)$ can lead to potential losses of precision due to cancellations of nearly equal numbers.

This simple example demonstrates the loss of numerical precision due to roundoff errors, where the number of leading digits is lost in a subtraction of two near equal numbers. The lesson

Calculating roots of quadratic equations!

Usual way of solving:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$

Potentially problematic if a or c (or both) are small leading to discriminant that is very close to the value of 'b'!

Correct way:

$$q \equiv -\frac{1}{2}\left[b + \text{sgn}(b)\sqrt{b^2 - 4ac}\right]$$

With roots calculated as:

$$x_1 = \frac{q}{a} \quad \text{and} \quad x_2 = \frac{c}{q}$$

Truncation errors

Round-off error is result of finite number of digits in our real representation and is characteristic of computer hardware.

Truncation error is a characteristic of practical implementation of (numerical) algorithm and it typically occurs as a consequence of applied "discretization" to otherwise continuous quantities, or when we limit infinite series to a finite number of terms.

TODO!

Numerical computing in modern C++

Has a long history, even though it only recently got value of PI properly in the C++ standard

Savior in 90ties as a successor to Fortan

Today it is somewhat "hidden behind the scene", in low level libraries used by fabulous Python tools

But when the going gets tough … it is still the best (and almost the only one) choice.

Many plans for C++ 26 standard

But, special functions have been part of the standard since C++17, and still Clang on MacOS doesn't have them.

Essential math headers

Cmath

Complex

Numbers

limits

numerical_limits<> class, and what it gives

But, C++ in its current standard gives only the basics.

Fortunately, there are great numerical libraries, for almost any domain or purpose, and two stand out in their generality and applicability.

GSL – GNU Scientific Library

Link - https://www.gnu.org/software/gsl/

Quoting:

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

Unlike the licenses of proprietary numerical libraries, the license of GSL does not restrict scientific cooperation. It allows you to share your programs freely with others.

Wide range of routines:

Complex Numbers	Quasi-Random Sequences	Discrete Hankel Transforms
Roots of Polynomials	Random Distributions	Root-Finding
Special Functions	Statistics	Minimization
Vectors and Matrices	Histograms	Least-Squares Fitting
Permutations	N-Tuples	Physical Constants
Sorting	Monte Carlo Integration	IEEE Floating-Point
BLAS Support	Simulated Annealing	Discrete Wavelet Transforms
Linear Algebra	Differential Equations	Basis splines
Eigensystems	Interpolation	Running Statistics
Fast Fourier Transforms	Numerical Differentiation	Sparse Matrices and Linear Algebra
Quadrature	Chebyshev Approximation	
Random Numbers	Series Acceleration	

C oriented!

Mostly you are working with pointers to C structs, allocated (mostly) on the heap.

Rich set of C-like manipulation routines for vectors, matrices, permutations, that usually look like this:

```
gsl_vector *gsl_vector_alloc(size_t n)
```

```
double gsl_vector_get(const gsl_vector *v, const size_t i)
```

```
int gsl_vector_add(gsl_vector *a, const gsl_vector *b)
```

```
gsl_matrix *gsl_matrix_alloc(size_t n1, size_t n2)
```

```
int gsl_matrix_transpose_memcpy(gsl_matrix *dest, const gsl_matrix *src)
```

Simple example of adaptive integration of following function, which has an algebraic-logarithmic singularity at the origin:

$$\int_0^1 x^{-1/2} \log(x) dx = -4$$

Looks like this:

```c
double f(double x, void* params) {
  double alpha = *(double*)params;
  double f = log(alpha * x) / sqrt(x);
  return f;
}

int gsl_integration(void)
{
  gsl_integration_workspace* w = gsl_integration_workspace_alloc(1000);

  double result, error;
  double expected = -4.0;
  double alpha = 1.0;

  gsl_function F;
  F.function = &f;
  F.params = &alpha;

  gsl_integration_qags(&F, 0, 1, 0, 1e-7, 1000, w, &result, &error);

  printf("result          = % .18f\n", result);
  printf("exact result    = % .18f\n", expected);
  printf("estimated error = % .18f\n", error);
  printf("actual error    = % .18f\n", result - expected);
  printf("intervals =  %d\n", w->size);

  return 0;
}
```

And example of solving Van der Pol differential equation

$$u''(t) + \mu u'(t)(u(t)^2 - 1) + u(t) = 0$$

When it is decoupled to two first order equations

$$u' = v$$
$$v' = -u + \mu v(1 - u^2)$$

Looks like this, where we define also Jacobian for this system.

```c
int func (double t, const double y[], double f[], void *params)
{
  (void)(t); /* avoid unused parameter warning */
  double mu = *(double *)params;

  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);

  return GSL_SUCCESS;
}
```

Jacobian

```
int jac (double t, const double y[], double *dfdy, double dfdt[], void *params)
{
  (void)(t); /* avoid unused parameter warning */
  double mu = *(double *)params;

  gsl_matrix_view dfdy_mat = gsl_matrix_view_array (dfdy, 2, 2);
  gsl_matrix *m = &dfdy_mat.matrix;

  gsl_matrix_set (m, 0, 0, 0.0);
  gsl_matrix_set (m, 0, 1, 1.0);
  gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
  gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));

  dfdt[0] = 0.0;
  dfdt[1] = 0.0;

  return GSL_SUCCESS;
}
```

Solving it with driver program, where we are using Prince-Dormand 8th order method to obtain solution (parameter 'gsl_odeiv2_step_rk8pd'):

```
void gsl_van_der_pol (void)
{
  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};

  gsl_odeiv2_driver * driver =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-6, 1e-6, 0.0);

  int i;
  double t = 0.0, t1 = 100.0;
  double y[2] = { 1.0, 0.0 };

  for (i = 1; i <= 100; i++)
    {
      double ti = i * t1 / 100.0;
      int status = gsl_odeiv2_driver_apply (driver, &t, ti, y);

      if (status != GSL_SUCCESS)
        {
          printf ("error, return value=%d\n", status);
          break;
        }

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_driver_free (driver);
}
```

Boost

Link - https://www.boost.org/

Premiere C++ library, if somewhat over-bloated (700+ Mb when extracted on hard disk).

Still, math functionality is mostly in standard Boost style, implemented solely in easily included headers … and there is much to include.

- Math library – special functions, polynomials, rational functions, root finding and function minimization, interpolation and numerical differentiation and integration

- Geometry – various geometry algorithms
- Multiprecision – support for __float128 type
- Odeint – solving differential equations
- QVM – quaternions, vectors, matrices
- uBLAS – linear algebra routines

Example of defining stiff Van der Pol system:

```
const double mu = 1000.0;

typedef boost::numeric::ublas::vector< double > vector_type;
typedef boost::numeric::ublas::matrix< double > matrix_type;

struct vdp_stiff
{
  void operator()(const vector_type& x, vector_type& dxdt, double t)
  {
    dxdt[0] = x[1];
    dxdt[1] = -x[0] - mu * x[1] * (x[0] * x[0] - 1.0);
  }
};

struct vdp_stiff_jacobi
{
  void operator()(const vector_type& x, matrix_type& J, const double& t, vector_type& dfdt)
  {
    J(0, 0) = 0.0;
    J(0, 1) = 1.0;
    J(1, 0) = -1.0 - 2.0 * mu * x[0] * x[1];
    J(1, 1) = -mu * (x[0] * x[0] - 1.0);

    dfdt[0] = 0.0;
    dfdt[1] = 0.0;
  }
};
```

And solving it using Rosenbrock method:

```
void boost_van_der_pol_stiff()
{
  vector_type x(2);
  srand(time(NULL));

  // initial conditions - set to random
  for (int i = 0; i < 2; i++)
    x[i] = (1.0 * rand()) / RAND_MAX;

  size_t num_of_steps = integrate_const(make_dense_output< rosenbrock4< double > >(1.0e-6, 1.0e-6),
                                        make_pair(vdp_stiff(), vdp_stiff_jacobi()),
                                        x, 0.0, 1000.0, 1.0,
                                        cout << phoenix::arg_names::arg2 << " "
                                             << phoenix::arg_names::arg1[0] << " "
                                             << phoenix::arg_names::arg1[1] << "\n"
                                       );

  clog << num_of_steps << endl;
}
```

Overview of other C++ numerical libraries of note

There are many, many, many

Wikipedia page gives good overview - LINK

Some of those of a more “general” orientation and applicability are:

Eigen - https://eigen.tuxfamily.org/

Armadillo - https://arma.sourceforge.net/

ALGLIB - https://www.alglib.net/

Blaze - https://bitbucket.org/blaze-lib

Trilinos - https://trilinos.github.io/

STARTING OUR OWN MINIMAL MATH LIBRARY

Why not just use Boost or GSL?

- We will be using them … in certain situations
- Learning by implementing and starting from the basics
- Creating general math library that is versatile and easily usable for physical simulations

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, but it should be quite easy customizing everything for C++14 and C++17.

MMLBase.h – setting the foundation

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, and we start with some global paths for our visualizers.

```
namespace MML
{
  // Global paths for library
  static const std::string MML_GLOBAL_PATH = "E:/Projects/MinimalMathLibrary";
  //static const std::string MML_GLOBAL_PATH = "/usr/zvanjak/projects/MinimalMathLibrary";

  static const std::string MML_PATH_ResultFiles = MML_GLOBAL_PATH + "/results/";

  static const std::string MML_PATH_RealFuncViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/real_function_visualizer/MML_RealFunctionVisualizer.exe";

  static const std::string MML_PATH_SurfaceViz_Win = MML_GLOBAL_PATH +
             "/tools/visualizers/win/scalar_function_2d_visualizer/MML_ScalarFunction2D_Visualizer.exe";

  static const std::string MML_PATH_ParametricCurve2DViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/parametric_curve_2d_visualizer/MML_ParametricCurve2D_Visualizer.exe";
  static const std::string MML_PATH_ParametricCurve3DViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/parametric_curve_3d_visualizer/MML_ParametricCurve3D_Visualizer.exe";

  static const std::string MML_PATH_VectorField2DViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/vector_field_2d_visualizer/MML_VectorField2D_Visualizer.exe";
  static const std::string MML_PATH_VectorField3DViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/vector_field_3d_visualizer/MML_VectorField3D_Visualizer.exe";

  static const std::string MML_PATH_Particle2DViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/particle_2d_visualizer/MML_ParticleVisualizer2D.exe";
  static const std::string MML_PATH_Particle3DViz_Win = MML_GLOBAL_PATH +
              "/tools/visualizers/win/particle_3d_visualizer/MML_ParticleVisualizer3D.exe";
```

Continuing with some basics

```
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));
}
```

Solving “inconvenience” that is a missing power operator in C++.

```
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; }
```

Solvers for quadratic, cubic and quartic equations.

```
// 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) { ... }
// Solving cubic equation a * x^3 + b * x^2 + c * x + d = 0
static int  SolveCubic(Real a, Real b, Real c, Real d,
                       Complex& x1, Complex& x2, Complex& x3) { ... }
// FIX - Solving quartic equation a * x^4 + b * x^3 + c * x^2 + d * x + e = 0
static void SolveQuartic(Real a, Real b, Real c, Real d, Real e,
                         Complex& x1, Complex& x2, Complex& x3, Complex& x4) { ... }
```

Constants

```
namespace Constants
{
  static inline const Real PI = std::numbers::pi;
  static inline const Real INV_PI = std::numbers::inv_pi;
  static inline const Real INV_SQRTPI = std::numbers::inv_sqrtpi;

  static inline const Real E = std::numbers::e;
  static inline const Real LN2 = std::numbers::ln2;
  static inline const Real LN10 = std::numbers::ln10;

  static inline const Real SQRT2 = std::numbers::sqrt2;
  static inline const Real SQRT3 = std::numbers::sqrt3;

  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();
}
```

In Defaults namespace we first define some default values for objects output

TODO – expand and improve

```
namespace Defaults
{
  // Output defaults
  static int VectorPrintWidth = 15;
  static int VectorPrintPrecision = 10;
  static int VectorNPrintWidth = 15;
  static int VectorNPrintPrecision = 10;
```

Then we define some defaults for precision of numerical operations.

```
//////////              Default precisions             ///////////
// Use the precision values based on the Real type
static inline const Real ComplexAreEqualTolerance    = PrecisionValues<Real>::ComplexAreEqualTolerance;
static inline const Real ComplexAreEqualAbsTolerance = PrecisionValues<Real>::ComplexAreEqualAbsTolerance;
static inline const Real VectorIsEqualTolerance      = PrecisionValues<Real>::VectorIsEqualTolerance;
static inline const Real MatrixIsEqualTolerance      = PrecisionValues<Real>::MatrixIsEqualTolerance;

static inline const Real Pnt2CartIsEqualTolerance  = PrecisionValues<Real>::Pnt2CartIsEqualTolerance;
static inline const Real Pnt2PolarIsEqualTolerance = PrecisionValues<Real>::Pnt2PolarIsEqualTolerance;
static inline const Real Pnt3CartIsEqualTolerance  = PrecisionValues<Real>::Pnt3CartIsEqualTolerance;
static inline const Real Pnt3SphIsEqualTolerance   = PrecisionValues<Real>::Pnt3SphIsEqualTolerance;
static inline const Real Pnt3CylIsEqualTolerance   = PrecisionValues<Real>::Pnt3CylIsEqualTolerance;

static inline const Real Vec2CartIsEqualTolerance    = PrecisionValues<Real>::Vec2CartIsEqualTolerance;
static inline const Real Vec3CartIsEqualTolerance    = PrecisionValues<Real>::Vec3CartIsEqualTolerance;
static inline const Real Vec3CartIsParallelTolerance = PrecisionValues<Real>::Vec3CartIsParallelTolerance;

static inline const Real Line3DIsPerpendicularTolerance = PrecisionValues<Real>::Line3DIsPerpendicularTolerance;
static inline const Real Line3DIsParallelTolerance      = PrecisionValues<Real>::Line3DIsParallelTolerance;
static inline const Real Plane3DIsPointOnPlaneTolerance = PrecisionValues<Real>::Plane3DIsPointOnPlaneTolerance;

static inline const Real IsMatrixSymmetricTolerance  = PrecisionValues<Real>::IsMatrixSymmetricTolerance;
static inline const Real IsMatrixDiagonalTolerance   = PrecisionValues<Real>::IsMatrixDiagonalTolerance;
static inline const Real IsMatrixUnitTolerance       = PrecisionValues<Real>::IsMatrixUnitTolerance;
static inline const Real IsMatrixOrthogonalTolerance = PrecisionValues<Real>::IsMatrixOrthogonalTolerance;
```

Customized for different ‘Real’ types with the use of template specialization.

General template.

```
// Template struct for precision values
template<typename T>
struct PrecisionValues;
```

Specialization for float type:

```
template<>
struct PrecisionValues<float>
{
    static constexpr float ComplexAreEqualTolerance = 1e-6f;
    static constexpr float ComplexAreEqualAbsTolerance = 1e-6f;

    static constexpr float MatrixIsEqualTolerance = 1e-6f;
    static constexpr float VectorIsEqualTolerance = 1e-6f;
```

And likewise (with different values) for double and long double types.

We will also specify a list of all standard math functions, available in almost every C++ compiler today

```
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); }
```

And a list of special functions, available from in C++17 standard (but still lacking in some compilers).

MSVC & gcc handle these correctly, but Clang is lacking, or to be more precise, Clang on MacOS (TODO – investigate properly)

```
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 – handling errors

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

No deep inheritance hierarchies.

Each exception is inherited from C++ standard exception that mostly resembles the problem and tries to report as much information as possible about possible causes.

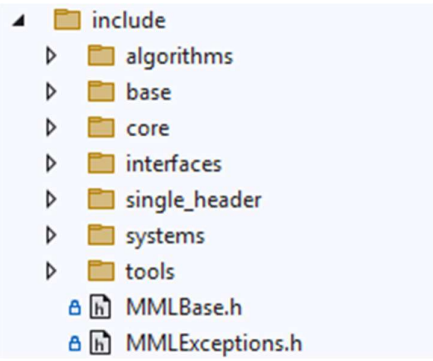
Set of classes defined in MMLExceptions.h

```
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 { ... };
}
```

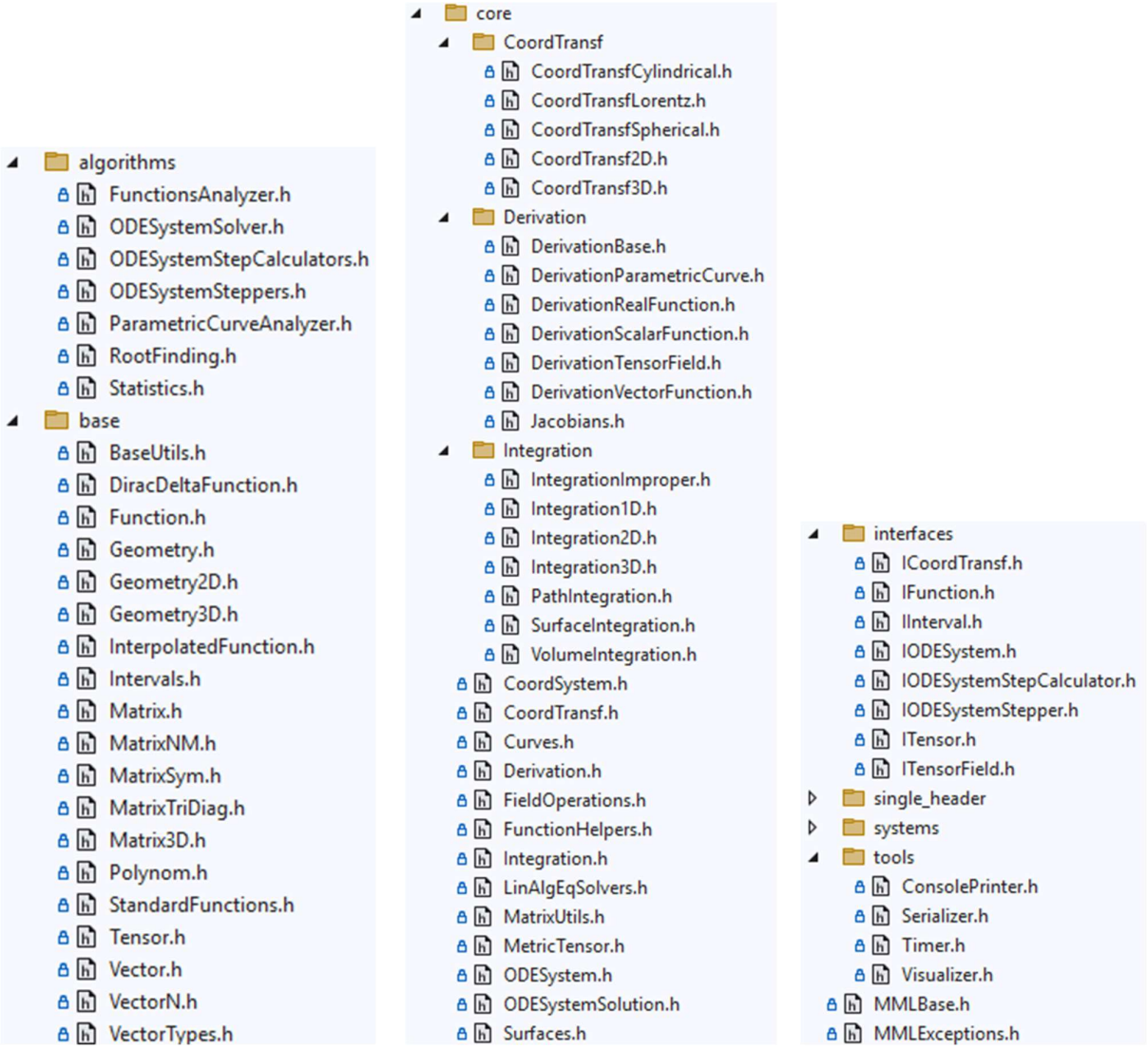
Code organization

Code is organized in three (+1) layers:

1) Base
2) Core
3) Algorithms
4) Tools - visualizers



Detailed preview.



Running code and examples

All code presented in this book is 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, Ubuntu, 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

Basic set of standard constructors.

```
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 Utils namespace

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

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

```
//////////////////                    Vector helpers                    //////////////////
static bool AreEqual(const Vector<Real> &a, const Vector<Real> &b,
                     Real eps = Defaults::VectorIsEqualTolerance) { ... }
static bool AreEqual(const Vector<Complex>& a, const Vector<Complex>& b,
                     Real eps = Defaults::ComplexAreEqualTolerance) { ... }
static bool AreEqualAbs(const Vector<Complex>& a, const Vector<Complex>& b,
                        Real eps = Defaults::ComplexAreEqualAbsTolerance) { ... }

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

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) { ... }

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

Example with vectors and its operations.

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

```
// REAL vectors
Vector<Real>    vec1(5);                    // init vector with 5 elements
Vector<Real>    vec2(3, 3.14159);           // init with constant value
Vector<Real>    vec3({ 1.5, -2.1, 0.48 });  // init with list of values

// using defined aliases (typedefs)
VectorDbl   vec4(vec3);               // init with copy ctor
VecD        vec5 = vec2;              // init with assignment

// initializing from std::vector and C/C++ array
std::vector<Real> std_vec{ -1.0, 5.0, -2.0 };
float  arr[5] = { -1.0, 5.0, -2.0, 10.0, 4.0 };

Vector<Real>    vec6(std_vec);             // init with std::vector<>
VectorFlt       vec7(5, arr);              // init with C/C++ array

// COMPLEX vectors
Vector<Complex>  vec_c1({ 1.0, 2.0, 3.0 });     // init with list of real values
VecC             vec_c2({ Complex(1,1),
                          Complex(-1,2),
                          Complex(2, -0.5) });  // init with list of complex values
```

```
// vector operations
Vector<Real> v1 = vec2 / 2.0 - 1.5 * vec6;
v1 /= vec4.NormL2();

Vector<Complex> v2 = vec_c2 * Complex(0.5, -1.5) + Utils::AddVec(v1, vec_c1) + Utils::MulVec(Complex(0, 1), v1);

Real    scalar_prod = Utils::ScalarProduct(vec2, vec3);
Complex scalar_prod_cmplx = Utils::ScalarProduct(vec_c1, vec_c2);
```

I/O

```
std::cout << "Real vectors:" << std::endl;
std::cout << "v1 = " << v1 << std::endl;
std::cout << "v1 = " << v1.to_string(10, 5) << std::endl;
std::cout << "v1 = "; v1.Print(std::cout, 7, 5); std::cout << std::endl;

std::cout << "\nComplex vectors:" << std::endl;
std::cout << "v2 = " << v2 << std::endl;
std::cout << "v2 = " << v2.to_string(10, 6) << std::endl;
std::cout << "v2 = "; v2.Print(std::cout, 10, 3); std::cout << std::endl;
```

Output

```
Real vectors:
v1 = [    1.169845766,     -2.25878164,     1.741283667]
v1 = [    1.1698,    -2.2588,     1.7413]
v1 = [ 1.1698, -2.2588,  1.7413]

Complex vectors:
v2 = [(4.169845766,0.1698457663), (2.24121836,0.2412183597), (4.991283667,-1.508716333)]
v2 = [(4.16985,0.169846), (2.24122,0.241218), (4.99128,-1.50872)]
v2 = [(4.17,0.17), (2.24,0.241), (4.99,-1.51)]
```

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

Data and private parts.

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

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

Set of available constructors.

```
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() { ... }
```

Some basic operations

```
void Resize(int rows, int cols, bool preserveElements = false) { ... }

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

Some more basics stuff, and some manipulation routines.

```
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 InitRowWithVector(int rowInd, const Vector<Type>& vec) { ... }
void InitColWithVector(int colInd, const Vector<Type>& vec) { ... }

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

Functions for calculating various matrix properties.

TODO – separate Norm calculations, and give math definitions

```
//////////////////////              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 the other one where you can specify wanted 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 Utils namespace with matrix helpers

Similarly to Vector helpers, we add to our BaseUtils.h header various helpers 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

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

Initializing matrices … in different ways.

By default, if elements are not initialized, they are set to 0.

```
Matrix<Real> a, b(2, 2), c(2, 2, { 1.0, -2.0, 3.75, -1.0 });
Matrix<Real> d(c), e = c;

// different types
MatrixInt      mat_int(2, 2, { 1, 2, 3, 4 });
MatrixComplex  mat_cmplx(2, 2, { Complex(1,1), Complex(-1,2),
                                 Complex(2,-0.5), Complex(1,1) });
```

We can extract sub-matrices

```
// initializing matrix with submatrix from existing matrix
Matrix<Real> f(5, 5, { 1,  2,  3,  4,  5,
                       6,  7,  8,  9,  10,
                       11, 12, 13, 14, 15,
                       16, 17, 18, 19, 20,
                       21, 22, 23, 24, 25 });
Matrix<Real> g(f, 3, 1, 2, 2);

// or using GetSubmatrix function()
g = f.GetSubmatrix(3, 1, 2, 2);
```

Creating special types of matrices

```
Matrix<Real>    unit_r = Matrix<Real>::GetUnitMatrix(3);
Matrix<Complex> unit_c = Matrix<Complex>::GetUnitMatrix(2);

Vector<Real> vec_a({ 1, 2, 3, 4, 5 });
Matrix<Real> diag = Utils::DiagonalMatrixFromVector(vec_a);
Matrix<Real> matA = Utils::RowMatrixFromVector<Real>(vec_a);
Matrix<Real> matB = Utils::ColumnMatrixFromVector<Real>(vec_a);
```

To illustrate matrix operations, we will implement Faddeev-Verrier algorithm for calculation of matrix characteristic polynomi.

```
Matrix<Real> A(3, 3, { 3, 1, 5,
                       3, 3, 1,
                       4, 6, 4 });

int N = A.RowNum();
Matrix<Real> M0(N, N);    // null matrix
Matrix<Real> I = Matrix<Real>::GetUnitMatrix(N);

Vector<Matrix<Real>> M(N + 1, M0);
RealPolynom c(N);
```

Algorithm:

```
M[0] = M0;
c[N] = 1.0;
for (int k = 1; k <= A.RowNum(); k++)
{
  M[k] = A * M[k - 1] + c[N - k + 1] * I;
  c[N - k] = -1.0 / k  * (A * M[k]).Trace();
}
Real det = std::pow(-1.0, N) * c[0];
Matrix<Real> Ainv = M[N] / det; // inverse of A

std::cout << "Characteristic polynomial: " << c << std::endl;
std::cout << "Determinant of A: " << det << std::endl;
std::cout << "Inverse of A: " << Ainv << std::endl;
```

Output:

```
M[0] = Rows: 3 Cols: 3
[          0,          0,          0 ]
[          0,          0,          0 ]
[          0,          0,          0 ]
c[3] = 1
M[1] = Rows: 3 Cols: 3
[          1,          0,          0 ]
[          0,          1,          0 ]
[          0,          0,          1 ]
c[2] = -10
M[2] = Rows: 3 Cols: 3
[         -7,          1,          5 ]
[          3,         -7,          1 ]
[          4,          6,         -6 ]
c[1] = 4
M[3] = Rows: 3 Cols: 3
[          6,         26,        -14 ]
[         -8,         -8,         12 ]
[          6,        -14,          6 ]
c[0] = -40
```

```
Characteristic polynomial: x^3 - 10*x^2 + 4*x - 40
Determinant of A: 40
Inverse of A: Rows: 3 Cols: 3
[       0.15,       0.65,      -0.35 ]
[       -0.2,       -0.2,        0.3 ]
[       0.15,      -0.35,       0.15 ]
A * Ainv = Rows: 3 Cols: 3
[          1,   2.22e-16,   2.22e-16 ]
[  -1.39e-16,          1,   8.33e-17 ]
[  -2.22e-16,          0,          1 ]
```

TENSORS

What is a tensor?

Basic interface for rank 2 tensor (similarly defined for tensors up to the 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) { ... }
};
```

Example usage

Defining tensors.

```
Tensor2<2> t2(2, 0, { 1.0, 2.0,
                      3.0, 4.0 });

Tensor2<3> t2_3(1, 1, { 1.0, 2.0, 2.0,
                        3.0, 4.0, 0.5,
                        5.0, 6.0, 1.5 });

Tensor2<4> t2_4(2, 0, {-1.0, 0.0, 0.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        0.0, 0.0, 1.0, 0.0,
                        0.0, 0.0, 0.0, 1.0 });

Tensor3<3> t3(1, 2, { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
                      10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0,
                      19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0 });

std::cout << "t2   = " << t2 << std::endl;
std::cout << "t2_3 = " << t2_3 << std::endl;
std::cout << "t2_4 = " << t2_4 << std::endl;
```

Operations with tensors

```cpp
// arithmetic operations
Tensor2<3> t2_3_2 = t2_3 + t2_3;
Tensor2<3> t2_3_3 = t2_3 - t2_3;
Tensor2<3> t2_3_4 = t2_3 * 2.0;
Tensor2<3> t2_3_5 = t2_3 / 2.0;
Tensor2<3> t2_3_6 =  2.0 * t2_3;
```

```cpp
// contraction
Real val1 = t2_3.Contract();
std::cout << "t2_3.Contract() = " << val1 << std::endl;

VectorN<Real, 3> vec_c1 = t3.Contract(0, 1);
std::cout << "t3.Contract(0, 1) = " << vec_c1 << std::endl;
VectorN<Real, 3> vec_c2 = t3.Contract(0, 2);
std::cout << "t3.Contract(0, 2) = " << vec_c2 << std::endl;
```

```cpp
// second rank tensor operating on two vectors
VectorN<Real, 3> vec1{ 1.0, -2.0, 3.0 }, vec2{ 0.5, 1.0, -1.5 };
Real val2 = t2_3(vec1, vec2);
std::cout << "t2_3(vec1, vec2) = " << val2 << std::endl;

VectorN<Real, 4> vec3{ 1.0, -2.0, 3.0, 1.0 }, vec4{ 0.5, 1.0, -1.5, 1.0 };
Real val3 = t2_4(vec3, vec4);
std::cout << "t2_4(vec3, vec4) = " << val3 << std::endl;
```

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 rich 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() {}
};
```

For this interface, we provide two implementations in
https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/mml/base/Function.h

First one expects function pointer, meaning it will also accept lambda expression with function definition, giving the user an easy way of creating RealFunction objects:

```
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); }
};
```

Second one is initialized with std::function<> object, providing for more complex situations when creating RealFunction objects (for example, when you want to create RealFunction from member function in some class).

```
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); }
};
```

Simple example of creating real functions

```
Real already_existing_func(Real x) {
  return sin(x) * (1 + x * x / 2);
}

void Real_functions_case_1_usage()
{
  // creating a function object from an already existing (standalone) function
  RealFunction fReal(already_existing_func);

  // we can also use lambda directly
  RealFunction fReal2( [](Real x) { return sin(x) * (1 + x * x / 2); } );
}
```

More examples, with various capabilities, including using member functions from already existing classes, are given at the end of this section.

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() {}
};
```

Similarly to RealFunction, it also provides two general implementations of interface.

```
//////////////////////////     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); }
};
```

TODO – simple example

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); }
};
```

In similar vein to VectorFunction<N>, we have VectorFunctionNM<N, M>, where input vector is of dimension N, and the function produces vector of different dimension M.

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

```cpp
// 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() {}
};
```

ITensorField<N>

Gives a tensor for every point in space.

Example for second rank tensor.

```cpp
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() {}
};
```

Example usage

We have five different possibilites for creating function objects.

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

CASE 1 - we already have standalone function (with expected declaration) providing needed calculation

For example, if we already have a defined set of existing functions:

```
Real Docs_Demo_functions_RealFunc(Real x) {
  return sin(x) * (1 + x * x / 2);
}

// simple scalar function, returning 1 / r, for given point
Real Docs_Demo_functions_ScalarFunc(const VectorN<Real, 3>& x) {
  return 1 / x.NormL2();
}

// little bit more realistic scalar function :)
Real Docs_Demo_functions_two_masses_gravity_potential(const VectorN<Real, 3>& x)
{
  const VectorN<Real, 3> x1{ 10.0, 0.0, 0.0 };
  const VectorN<Real, 3> x2{ -10.0, 0.0, 0.0 };
  const Real m1 = 1000.0;
  const Real m2 = 1000.0;
  const Real G = 1.0;
  return -G * m1 / (x - x1).NormL2() - G * m2 / (x - x2).NormL2();
}

// TODO - vector function returning field with curl in 2D
VectorN<Real, 2> Docs_Demo_functions_VectorFunc(const VectorN<Real, 2>& x) {
  return VectorN<Real, 2>{0, x[0] * x[1]};
}

// helix curve in 3D
VectorN<Real, 3> Docs_Demo_functions_ParamCurve(Real t) {
  return  VectorN<Real,3>{ cos(t), sin(t), t};
}

// simple surface
VectorN<Real, 3> Docs_Demo_functions_ParamSurface(Real x, Real y) {
  return VectorN<Real, 3>{x * y, 2 * x * y, 3 * x};
}
```

We can simply create appropriate functions objects by passing function pointers.

```
// creating a function objects from an already existing (standalone) function
RealFunction            fReal(Docs_Demo_functions_RealFunc);
ScalarFunction<3>       fScalar(Docs_Demo_functions_ScalarFunc);
ScalarFunction<3>       fScalar2(Docs_Demo_functions_two_masses_gravity_potential);
VectorFunction<2>       fVector(Docs_Demo_functions_VectorFunc);
ParametricCurve<3>      paramCurve(Docs_Demo_functions_ParamCurve);
ParametricSurface<3>    paramSurface(Docs_Demo_functions_ParamSurface);
```

CASE 2 - creating function directly from lambda

Extra convenient and useful!

```
RealFunction f2{ [](Real x) { return (Real)sin(x) * (1 + x * x / 2); } };

ScalarFunction<3> fScalar([](const VectorN<Real, 3>& x) { return 1 / x.NormL2(); });
ScalarFunction<3> two_masses_gravity_field_potential{ [](const VectorN<Real, 3>& x)
            {
                const VectorN<Real, 3> x1{ 10.0, 0.0, 0.0 };
                const VectorN<Real, 3> x2{ -10.0, 0.0, 0.0 };
                const Real m1 = 1000.0;
                const Real m2 = 1000.0;
                const Real G = 1.0;
                return -G * m1 / (x - x1).NormL2() - G * m2 / (x - x2).NormL2();
            }
};
VectorFunction<3>       fVector([](const VectorN<Real, 3>& x)
                                {
                                  return VectorN<Real, 3>{0, x[0] * x[1], 0};
                                });
ParametricCurve<3>      paramCurve([](Real t) { return VectorN<Real, 3>{t, 2 * t, 3 * t}; });
ParametricSurface<3>    paramSurface([](Real x, Real y) { return VectorN<Real, 3>{x* y, 2 * x * y, 3 * x}; });
```

CASE 3 - we have function that has some additional parameters that it depends on (that we might want to vary easily)

We create a class wrapper around that function(ality), and inherit it from appropriate function interface.

Taking as an example function for calculating gravity potential of two masses, where in preceding two examples we were forced to fix both masses and positions, we can do this:

```
class TwoMassesGravityPotential : public IScalarFunction<3>
{
  Real _m1, _m2, _G;
  VectorN<Real, 3> _x1, _x2;

public:
  TwoMassesGravityPotential(Real m1, Real m2, Real G,
                            const VectorN<Real, 3>& x1, const VectorN<Real, 3>& x2)
    : _m1(m1), _m2(m2), _G(G), _x1(x1), _x2(x2) {}

  void SetM1(Real m1) { _m1 = m1; }
  void SetM2(Real m2) { _m2 = m2; }
  void SetX1(const VectorN<Real, 3>& x1) { _x1 = x1; }
  void SetX2(const VectorN<Real, 3>& x2) { _x2 = x2; }

  Real operator()(const VectorN<Real, 3>& x) const
  {
    return -_G * (_m1 / (x - _x1).NormL2() + _m2 / (x - _x2).NormL2());
  }
};
```

And simply use objects constructed from TwoMassesGravityPotential class as scalar functions wherever needed.

CASE 4 - there is an already existing external class that has EXACT FUNCTION you want to use (and you can't change the class)

Big class

```
class BigComplexClassYouCantChange1 {
public:
  double _param1, _param2;
  double Weight(double x) const { return 58 * _param1; }

  double UsefulRealFunc(double x) const { return (_param1 * cos(x) + _param2 * sin(x)) / Weight(x); }
};
```

We create RealFunctionFromStdFunc

```
BigComplexClassYouCantChange1 bigObj;

// set parameter values as needed
bigObj._param1 = 1.0;
bigObj._param2 = 2.0;

// declaring f4 as a RealFunction object
RealFunctionFromStdFunc f4(std::function<Real(Real)>{ [&bigObj](Real x)
                                                        {
                                                          return bigObj.UsefulRealFunc(x);
                                                        }
                                                    }
                          );
```

And use f4 as normal RealFunction object.

CASE 5 - there is an external class that has data relevant to calculation for your real function, but you can't (or don’t want to) change the class.

Class

```
class BigComplexClassYouCantChange2 {
public:
  double _param1, _param2;
  double Weight(double x) const { return 58*_param1; }
};
```

Create a helper class, inherit it from IRealFunction, wrap relevant object with const reference, and use it as RealFunction object wherever it is needed

```
class BigComplexRealFunc2 : public IRealFunction {
  const BigComplexClassYouCantChange2& _ref;
public:
  BigComplexRealFunc2(const BigComplexClassYouCantChange2& bigClass) : _ref(bigClass) { }

  Real operator()(Real x) const {
    return (_ref._param1 * cos(x) + _ref._param2 * sin(x)) / _ref.Weight(x) ;
  }
};
```

Usage

```
BigComplexClassYouCantChange2 bigObj;

bigObj._param1 = 1.0;
bigObj._param2 = 2.0;

BigComplexRealFunc2    f5(bigObj);     // usable RealFunction object

std::cout << "f5(0.0) = " << f5(0.0) << std::endl;
std::cout << "f5(1.0) = " << f5(1.0) << std::endl;
std::cout << "f5(2.0) = " << f5(2.0) << std::endl;
```

CASE 6 - there is an external class that has data relevant to calculation and you need to define multiple real function based on that data

TODO - Show examples of creating different kinds of:

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

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"? In authors opinion, semantics is important, and type system should be utilized to support it.

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 10 – 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 our case, in both 2D and 3D cases they are specified with point, and vector representing direction, ie. representing parametric line equation.

Again, example in 3D

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() member functions … 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{ ... }

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

Triangle in 3D space

```
class Triangle3D
{
protected:
  Point3Cartesian _pnt1, _pnt2, _pnt3;
public:
  Triangle3D() { }
  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; }

  Real Area() const{ ... }

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

  Plane3D getDefinedPlane() const{ ... }
};
```

Making Triangle3D a ParametricSurface<3>

So it can be used in surface integration routines!

```
// Represents triangular surface in 3D and  defines all needed mappings
// to represent Triangle3D as IParametricSurface
class TriangleSurface3D : public Triangle3D, IParametricSurface<3>
{
public:
  Real _minX, _maxX, _minY, _maxY;
  Point3Cartesian _origin;
  Point3Cartesian _center;
  Vector3Cartesian _localX, _localY;
  Vector3Cartesian _normal;
  Real _pnt3XCoord;

  // pnt1-pnt2 should be hypothenuse of the triangle!!!
  // but we will handle it, if it is not
  TriangleSurface3D(Point3Cartesian pnt1, Point3Cartesian pnt2, Point3Cartesian pnt3){ ... }

  virtual Real getMinU() const { return _minX; }
  virtual Real getMaxU() const { return _maxX; }
  virtual Real getMinW(Real u) const { return _minY; }
  virtual Real getMaxW(Real u) const{ ... }

  // for a given (u,w), which is basically (x,y) in local coordinate system
  // return point in global coordinate system
  VectorN<Real, 3> operator()(Real u, Real w) const{ ... }
};
```

Boxes

TODO!

Modeling surfaces & solid bodies

Two approaches

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

```
class IBody
{
public:
  virtual bool isInside(const Point3Cartesian& pnt) const = 0;
};
```

```
class ISolidBodyWithBoundary : public IBody
{
protected:
  Real _x1, _x2;

  Real(*_y1)(Real);
  Real(*_y2)(Real);

  Real(*_z1)(Real, Real);
  Real(*_z2)(Real, Real);

public:
  ISolidBodyWithBoundary(Real x1, Real x2,
                         Real(*y1)(Real), Real(*y2)(Real),
                         Real(*z1)(Real, Real), Real(*z2)(Real, Real))
     : _x1(x1), _x2(x2), _y1(y1), _y2(y2), _z1(z1), _z2(z2)
  { }

  virtual Real getDensity(const VectorN<Real, 3>& x) const = 0;

  virtual bool isInside(const Point3Cartesian& pnt) const { ... }

};
```

Two implementations:

Constant density

```
class SolidBodyWithBoundaryConstDensity : public ISolidBodyWithBoundary
{
  Real _density;
public:
  SolidBodyWithBoundaryConstDensity(Real x1, Real x2, Real(*y1)(Real), Real(*y2)(Real),
                                    Real(*z1)(Real, Real), Real(*z2)(Real, Real),
                                    Real density)
    : ISolidBodyWithBoundary(x1, x2, y1, y2, z1, z2), _density(density)
  { }

  virtual Real getDensity(const VectorN<Real, 3>& x) const
  {
    return _density;
  }
};
```

Variable density

```
class SolidBodyWithBoundary : public ISolidBodyWithBoundary
{
  Real(*_density)(const VectorN<Real, 3>& x);
public:
  SolidBodyWithBoundary(Real x1, Real x2, Real(*y1)(Real), Real(*y2)(Real),
                        Real(*z1)(Real, Real), Real(*z2)(Real, Real),
                        Real(*density)(const VectorN<Real, 3>& x))
    : ISolidBodyWithBoundary(x1, x2, y1, y2, z1, z2), _density(density)
  { }

  virtual Real getDensity(const VectorN<Real, 3>& x) const
  {
    return _density(x);
  }
};
```

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. Visualization, anyone?

Two ways:

1) Handling visualization data directly in application
2) Exchanging visualization data through files

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 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 ON WINDOWS

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 distinct applications, with each one of them focusing on one type of visualization. So far, following visualizations are available:

1) RealFunctionVisualizer (with capability of visualizing multiple functions)
2) ScalarFunction2DVisualizer – for visualizing 2D surfaces given as z=f(x,y) on rectangular patch
3) ParametricCurve2DVisualizer
4) ParametricCurve3DVisualizer
5) VectorField2DVisualizer – for visualizing vector field in 2D
6) VectorField3DVisualizer – for
7) ParticleVisualizer2D
8) ParticleVisualizer3D

Helper Serializer class

To relieve user of manually creating input files for given visualizers, Serializer class was created, with an extensive set of static functions for writing to files different kind of data.

Functionality for serializing real functions.

```
// Real function serialization
// serializing values at equally spaced points in given interval
static bool SaveRealFunc(const IRealFunction& f, std::string title,
                         Real x1, Real x2, int numPoints, std::string fileName) { ... }

// serializing values at given list of points
static bool SaveRealFunc(const IRealFunction& f, std::string title,
                         Vector<Real> points, std::string fileName) { ... }

// serializing multiple functions in a single files
static bool SaveRealMultiFunc(std::vector<IRealFunction*> funcs, std::string title,
                              Real x1, Real x2, int numPoints, std::string fileName) { ... }

// same as SaveRealFunc, but points are not explicitly written in file
// (as they can be calculated fomr x1, x2 and numPoints)
static bool SaveRealFuncEquallySpaced(const IRealFunction& f, std::string title,
                                      Real x1, Real x2, int numPoints, std::string fileName)
```

Parametric curves

```
// 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 SaveAsParamCurve2D(Vector<Real> vec_x, Vector<Real> vec_y) { ... }

// Helper/forwarding functions
static bool SaveParamCurveCartesian2D(const IRealToVectorFunction<2>& f, 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) { ... }
```

Scalar function serialization

```
// 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

2D versions

```
// 2D vector function serialization
static bool SaveVectorFunc2D( const IVectorFunction<2>& f, std::string inType, std::string title,
                              Real x1_start, Real x1_end, int numPointsX1,
                              Real x2_start, Real x2_end, int numPointsX2, std::string fileName)

static bool SaveVectorFunc2D( const IVectorFunction<2>& f, std::string inType, std::string title,
                              Real x1_start, Real x1_end, int numPointsX1,
                              Real x2_start, Real x2_end, int numPointsX2,
                              std::string fileName, Real upper_threshold) { ... }

static bool SaveVectorFunc2DCartesian(const IVectorFunction<2>& f, std::string title,
                                      Real x1, Real x2, int numPointsX,
                                      Real y1, Real y2, int numPointsY, std::string fileName) { .

static bool SaveVectorFunc2DCartesian(const IVectorFunction<2>& f, std::string title,
                                      Real x1, Real x2, int numPointsX,
                                      Real y1, Real y2, int numPointsY, std::string fileName,
                                      Real upper_threshold) { ... }
```

And 3D versions

```
// 3D 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) { ... }
```

Particle simulation serializers

```
// particle simulation serialization
static bool SaveParticleSimulation2D(std::string fileName, int numBalls,
                                     std::vector<std::vector<Pnt2Cart>> ballPositions,
                                     std::vector<std::string> ballColors, std::vector<double> ballRadius )

static bool SaveParticleSimulation3D(std::string fileName, int numBalls,
                                     std::vector<std::vector<Pnt3Cart>> ballPositions,
                                     std::vector<std::string> ballColors, std::vector<double> ballRadius)
```

ODE solution serializers

```
// ODESolution serialization
static bool SaveODESolutionAsMultiFunc(const ODESystemSolution& sol, std::string title, std::string fileName)

static bool SaveODESolAsParametricCurve2D(const ODESystemSolution& sol, std::string fileName,
                                          int ind1, int ind2, std::string title) { ... }

static bool SaveODESolAsParametricCurve3D(const ODESystemSolution& sol, std::string fileName,
                                          int ind1, int ind2, int ind3, std::string title) { ... }
```

Helper Visualizer class

- All WPF visualizers are distinct applications, so from our console applications they have to be started with appropriate parameters, ie. names of input files
- Serializer helps in creating those files,
- Visualizer ties it all together

We are starting with a set of paths:

```
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 _pathParametricCurve3DViz{ MML_PATH_ParametricCurve3DViz };
  static inline std::string _pathParametricCurve2DViz{ MML_PATH_ParametricCurve2DViz };
  static inline std::string _pathVectorField2DViz{ MML_PATH_VectorField2DViz };
  static inline std::string _pathVectorField3DViz{ MML_PATH_VectorField3DViz };

  static inline std::string _pathParticle2DViz{ MML_PATH_Particle2DViz };
  static inline std::string _pathParticle3DViz{ MML_PATH_Particle3DViz };
```

Set of visualizers for real functions

```
// visualizations of Real function
static void VisualizeRealFunction(const IRealFunction& f, std::string title,
                                  Real x1, Real x2, int numPoints, std::string fileName) { ..

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

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

Visualization for scalar function in 2D.

```
// visualizations of Scalar function in 2D
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)
```

Visualizations for vector fields

```
// visualizations of Vector fields
static void VisualizeVectorField2DCartesian(const IVectorFunction<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)
```

Parametric curves have extensive support for visualization.

```
static void VisualizeParamCurve2D(const IRealToVectorFunction<2>& f, std::string title,
                                  Real t1, Real t2, int numPoints,
                                  std::string fileName) { ... }

static void VisualizeMultiParamCurve2D(std::vector<IRealToVectorFunction<2>*> curves,
                                       std::string title,
                                       Real t1, Real t2,
                                       int numPoints, 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<IRealToVectorFunction<3>*> curves,
                                       std::string title,
                                       Real t1, Real t2, int numPoints,
                                       std::string fileName) { ... }

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

ODE system solution visualizations.

```
// ODE Solution visualizations
static void VisualizeODESysSolAsMultiFunc(const ODESystemSolution& sol,
                                          std::string title, std::string fileName) {

static void VisualizeODESysSolAsParamCurve2(const ODESystemSolution& sol,
                                            int ind1, int ind2,
                                            std::string title, std::string fileName)

static void VisualizeODESysSolAsParamCurve3(const ODESystemSolution& sol,
                                            int ind1, int ind2, int ind3,
                                            std::string title, std::string fileName)

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

Example how it is mostly done in all functions (basically, it uses existing Serializer object that can serialize to a file value for different kinds of functions, and then, using defined paths for visualizers, creates system command to call them with appropriate parameters):

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

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 WPF Canvas.

Supported input types:

REAL_FUNCTION

Visualizing real function is quite simple:

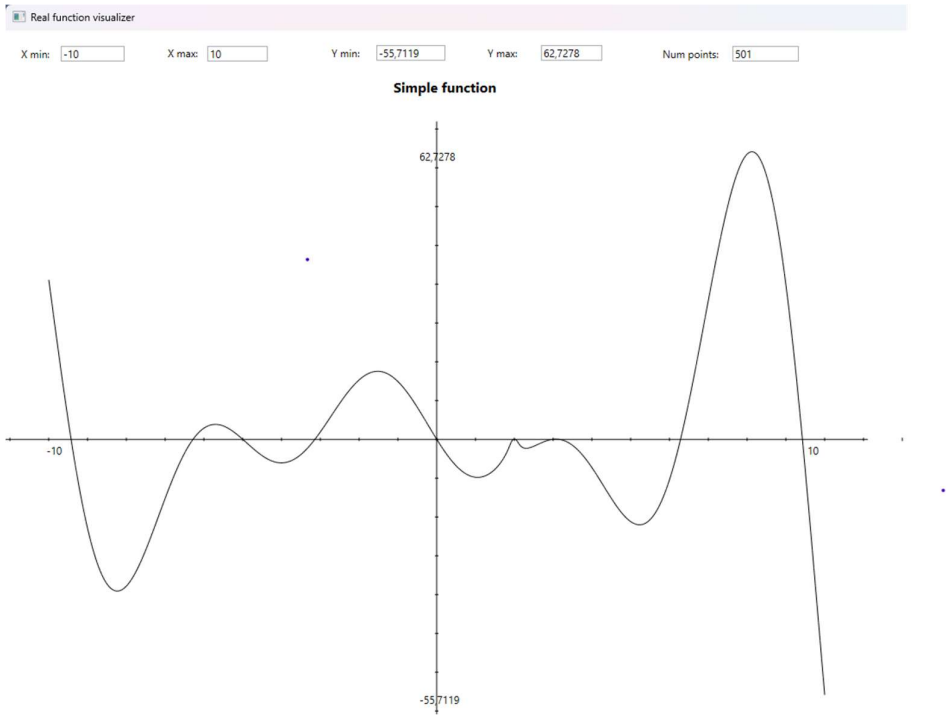
```
RealFunction f1{ [](Real x) { return sin(x) * (x - 3) * (x + 5) / exp(0.2 / std::abs(2 - x)); } };

Visualizer::VisualizeRealFunction(f1, "Simple function", -10.0, 10.0, 501,
                                  "example3_wpf_real_func1.txt");
```

File “example3_wpf_real_func.txt” has following format:

```
REAL_FUNCTION
Simple function
x1: -10
x2: 10
NumPoints: 501
-10 34.7769
-9.96 32.242
-9.92 29.7058
-9.88 27.1733
-9.84 24.6492
-9.8 22.1381
…
9.8 -35.9488
9.84 -39.9153
9.88 -43.8821
9.92 -47.842
```

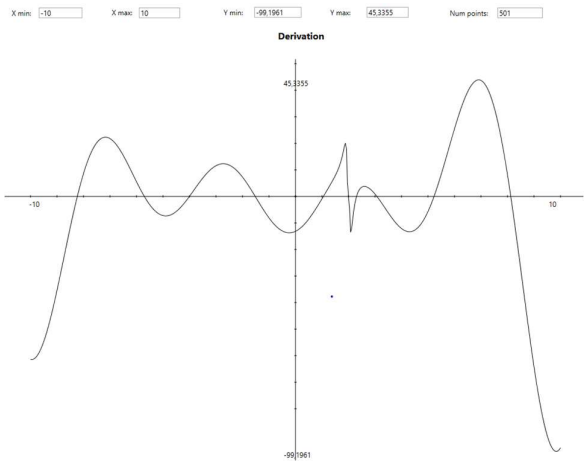
```
9.96 -51.7877
10 -55.7119
```

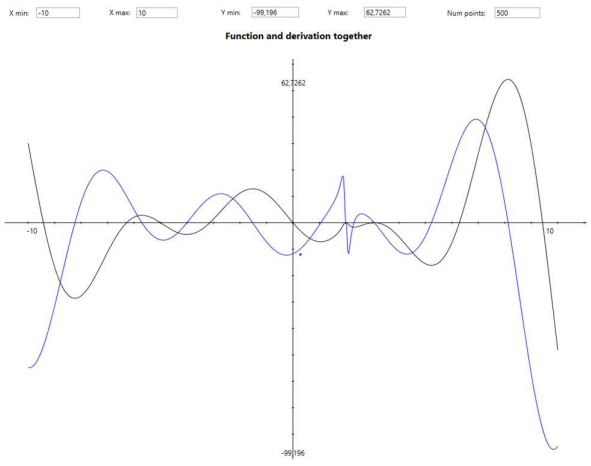
And we get following visualization:



We can easily calculate and then visualize function derivation (and show both together).

```
RealFuncDerived4 f1_der(f1);      // 4th order derivation of f1 function

Visualizer::VisualizeRealFunction(f1_der, "Derivation", -10.0, 10.0, 501,
                                  "example3_wpf_real_func2.txt");

// shown together
Visualizer::VisualizeMultiRealFunction({ &f1, &f1_der }, "Function and derivation together",
                                -10.0, 10.0, 501, "example3_wpf_multi_real_func.txt");
```





REAL_FUNCTION_EQUALLY_SPACED

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

Use if you need to save on file size.

MULTI_REAL_FUNCTION

In previous example we’ve seen how we can visualize real functions when we have different object representing them, but one other important usage is for visualizing solutions of differential equations.

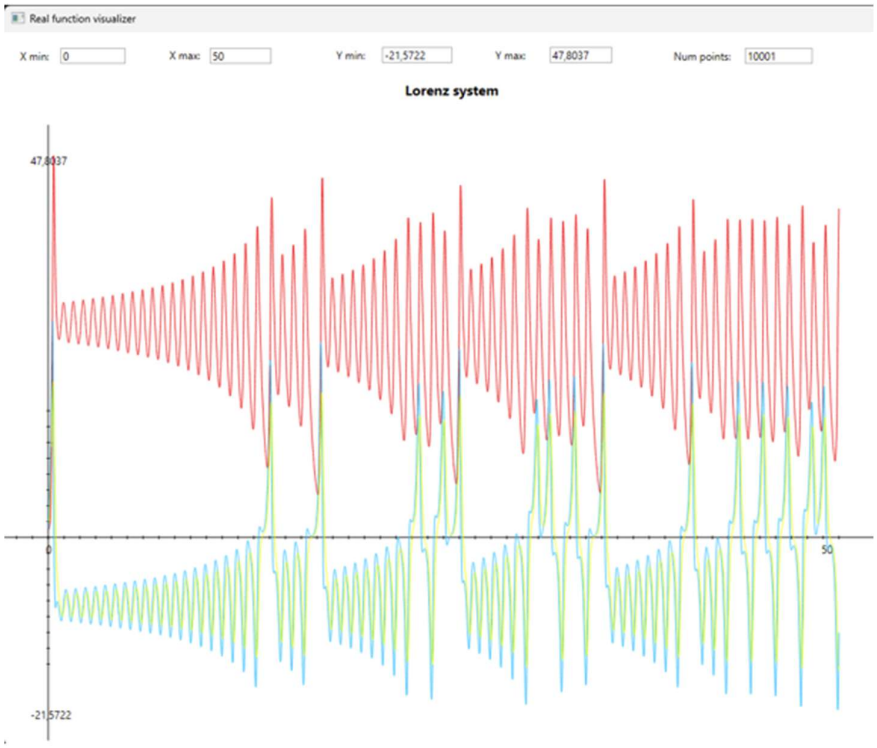
Solving Lorentz system is simple (OK, we are getting a little bit ahead of ourselves):

```
// Solving Lorentz system as MULTI_REAL_FUNCTION example
ODESystem lorentz_system(3, [](Real t, const Vector<Real>& x, Vector<Real>& dxdt)
  {
    const Real sigma = 10.0;
    const Real rho = 28.0;
    const Real beta = 8.0 / 3.0;
    dxdt[0] = sigma * (x[1] - x[0]);
    dxdt[1] = x[0] * (rho - x[2]) - x[1];
    dxdt[2] = x[0] * x[1] - beta * x[2];
  }
);

Vector<Real> ystart({ 2.0, 1.0, 1.0 });
ODESystemSolver<RK5_CashKarp_Stepper> solver(lorentz_system);
ODESystemSolution sol = solver.integrate(ystart, 0.0, 50.0, 0.1, 1e-08, 0.01);

Visualizer::VisualizeODESysSolAsMultiFunc(sol, "Lorentz system", "example3_wpf_lorentz.txt");
```

We get following data in file (parameters at the beginning are: graph title, number of functions (expected number of values in the row), x1, x2 and number of points).

```
MULTI_REAL_FUNCTION
Lorentz system
3
x1: 0
x2: 50
NumPoints: 478
0 2 1 1
0.108108 3.65659 7.54213 1.7794
0.209965 10.8358 21.4095 10.6551
0.314271 19.4985 17.8098 44.4869
0.416696 6.86177 -7.84692 38.1623
0.517103 -2.91685 -8.29004 28.4874
0.620594 -6.26595 -8.2665 25.2455
…
49.4665 3.54238 -2.88674 30.0554
49.5697 -0.504479 -2.62469 22.5465
49.6769 -2.1692 -3.61403 17.3492
49.7775 -4.14473 -6.89177 14.7035
49.8823 -8.39311 -13.8369 17.0854
49.9881 -14.0495 -17.2098 30.9598
50 -14.3586 -16.3486 32.7962
```

And the following visualization.



Visualizing parametric curves in 2D and 3D

Parametric curves in 2D

We'll be using predefined curves.

Starting with lemniscate.

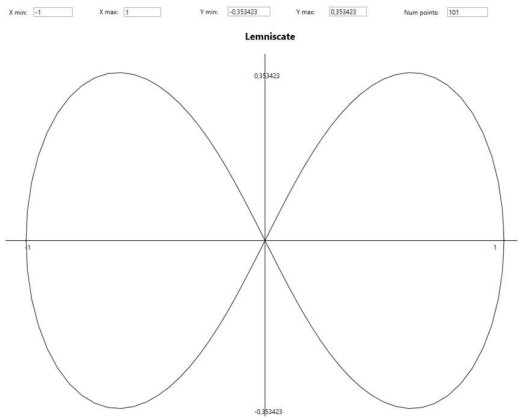
```
LemniscateCurve lemniscate;
Visualizer::VisualizeParamCurve2D(lemniscate, "Lemniscate", 0.0, 2 * Constants::PI, 101,
                                  "example3_wpf_curve_lemniscate.txt");
```

Generated file:

```
PARAMETRIC_CURVE_CARTESIAN_2D
Lemniscate
t1: 0
t2: 6.28319
NumPoints: 101
0 1 0
0.0628319 0.994107 0.0624205
0.125664 0.976771 0.122422
0.188496 0.948967 0.177819
0.251327 0.912169 0.226847
0.314159 0.868155 0.268275
```

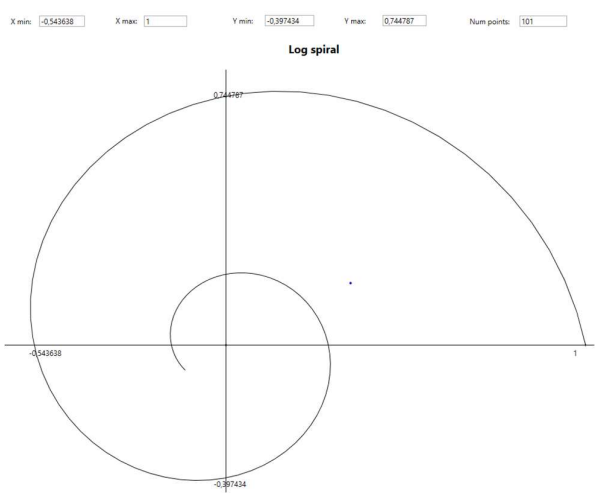
```
…
5.96903 0.868155 -0.268275
6.03186 0.912169 -0.226847
6.09469 0.948967 -0.177819
6.15752 0.976771 -0.122422
6.22035 0.994107 -0.0624205
6.28319 1 -9.12671e-15
```

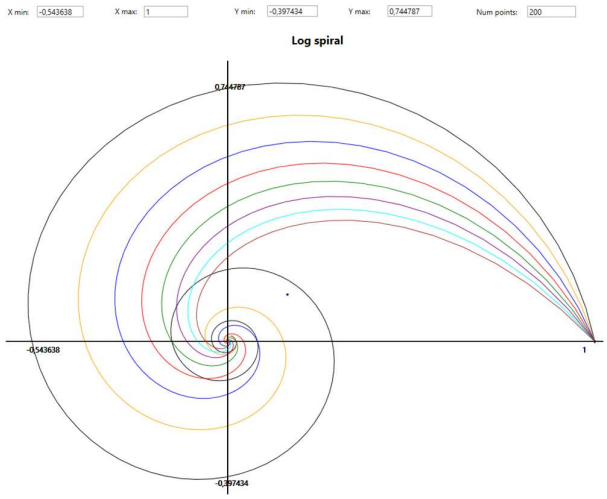
And visualization.



Log spiral curve

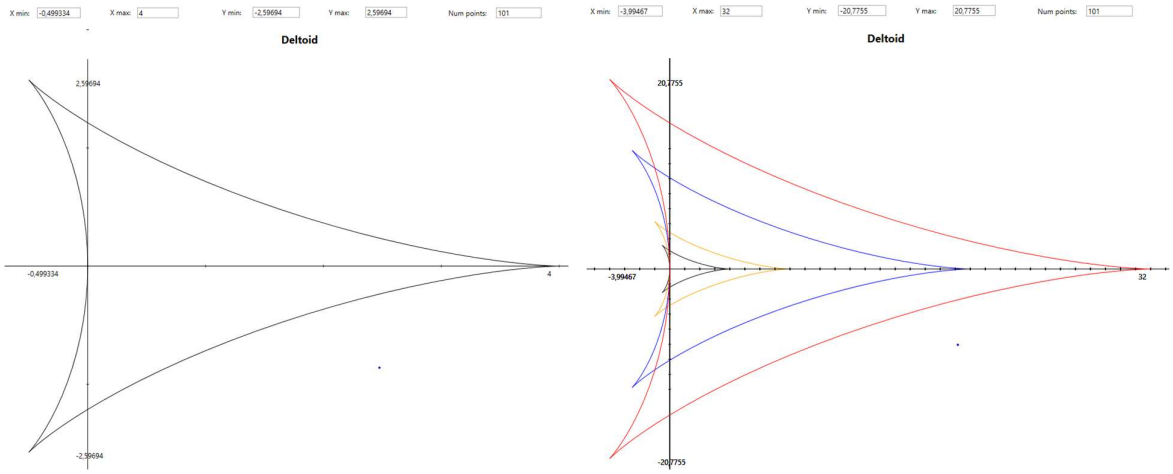
```
LogSpiralCurve  log_spiral2(-0.2), log_spiral3(-0.3), log_spiral4(-0.4), log_spiral5(-0.5), log_spiral6(-0.6);
LogSpiralCurve  log_spiral7(-0.7), log_spiral8(-0.8), log_spiral9(-0.9);

Visualizer::VisualizeParamCurve2D(log_spiral2, "Log spiral", 0.0, 10.0, 101,
                                  "example3_wpf_curve_log_spiral.txt");

Visualizer::VisualizeMultiParamCurve2D({ &log_spiral2, &log_spiral3, &log_spiral4, &log_spiral5,
                                         &log_spiral6, &log_spiral7, &log_spiral8, &log_spiral9 },
                                         "Log spiral", 0.0, 20.0, 201, "example3_wpf_curve_log_spiral_multi");
```





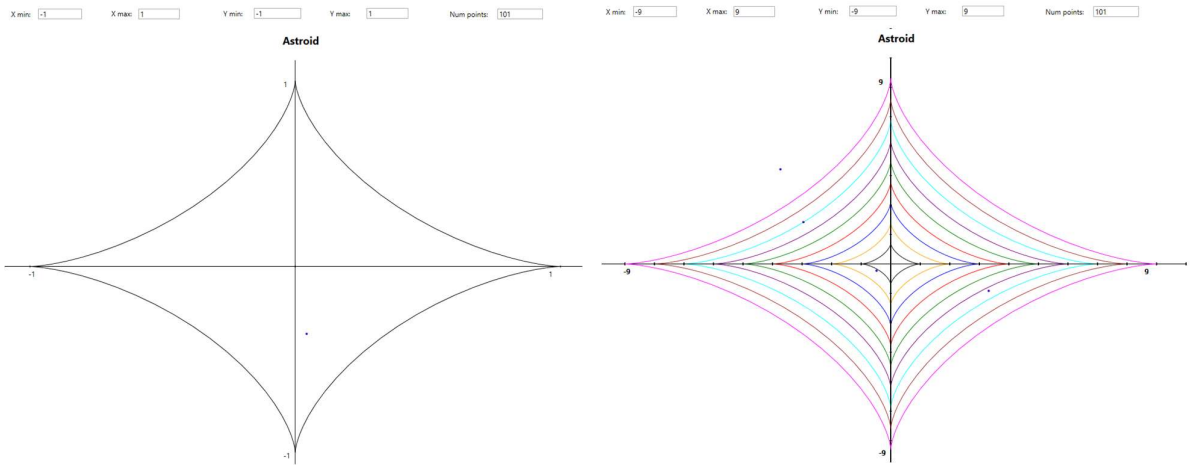
Deltoid

```
DeltoidCurve deltoid1(1), deltoid2(2), deltoid3(5), deltoid4(8), deltoid5(10), deltoid6(15);

Visualizer::VisualizeParamCurve2D(deltoid1, "Deltoid", 0.0, 2 * Constants::PI, 101,
                                  "example3_wpf_curve_deltoid.txt");

Visualizer::VisualizeMultiParamCurve2D({ &deltoid1, &deltoid2, &deltoid3, &deltoid4 },
                                       "Deltoid", 0.0, 2 * Constants::PI, 101,
                                       "example3_wpf_curve_deltoid_multi");
```



Astroid

```
AstroidCurve astroid1(1), astroid2(2.0), astroid3(3.0), astroid4(4.0);
AstroidCurve astroid5(5.0), astroid6(6.0), astroid7(7.0), astroid8(8.0), astroid9(9.0);

Visualizer::VisualizeParamCurve2D(astroid1, "Astroid", 0.0, 2 * Constants::PI, 101,
                                  "example3_wpf_curve_astroid.txt");

Visualizer::VisualizeMultiParamCurve2D({ &astroid1, &astroid2, &astroid3, &astroid4, &astroid5,
                                         &astroid6, &astroid7, &astroid8, &astroid9 },
                                       "Astroid", 0.0, 2 * Constants::PI, 101,
                                       "example3_wpf_curve_astroid_multi");
```

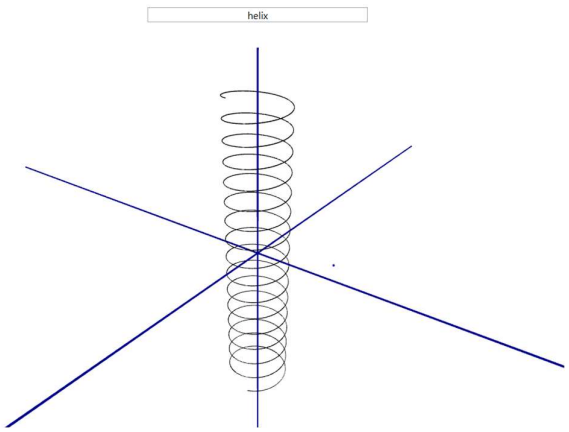


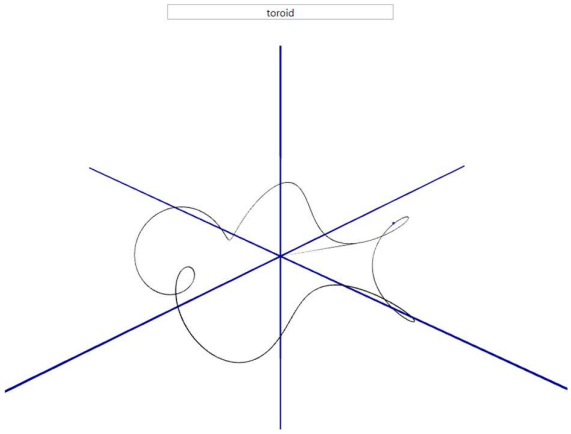
Parametric curves in 3D

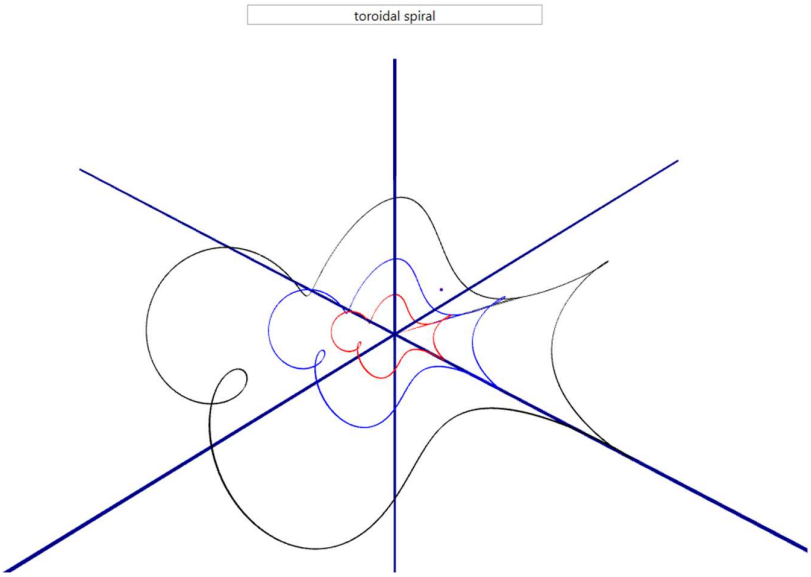
```
// using predefined 3D curves for visualization example
HelixCurve          helix(20.0, 2.0);
ToroidalSpiralCurve toroid1(5, 20.0), toroid2(5, 10.0), toroid3(5, 5.0);

Visualizer::VisualizeParamCurve3D(helix, "helix", -50.0, 50.0, 1000,
                                  "example3_wpf_curve_helix.txt");

Visualizer::VisualizeParamCurve3D(toroid1, "toroid", 0.0, 5 * Constants::PI, 1000,
                                  "example3_wpf_curve_toroid.txt");

// visualize all thre curves
Visualizer::VisualizeMultiParamCurve3D({ &toroid1, &toroid2, &toroid3 }, "toroidal spiral",
                                       0.0, 5 * Constants::PI, 1000,
                                       "example3_wpf_curve_toroidal_spiral.txt");
```

Generated content of file “example3_wpf_curve_helix.txt”:

```
PARAMETRIC_CURVE_CARTESIAN_3D
helix
t1: -50
t2: 50
NumPoints: 1000
-50 19.2993 5.2475 -100
-49.8999 18.6783 7.14987 -99.7998
-49.7998 17.8703 8.98066 -99.5996
-49.6997 16.8834 10.7215 -99.3994
-49.5996 15.7274 12.3551 -99.1992
…
49.4995 14.414 -13.8649 98.999
49.5996 15.7274 -12.3551 99.1992
49.6997 16.8834 -10.7215 99.3994
49.7998 17.8703 -8.98066 99.5996
49.8999 18.6783 -7.14987 99.7998
50 19.2993 -5.2475 100
```

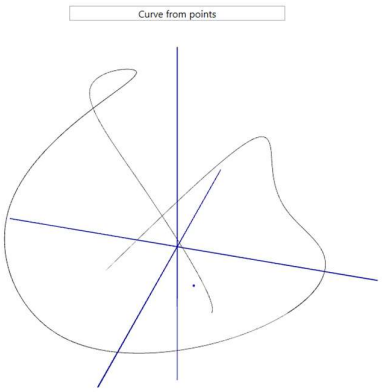


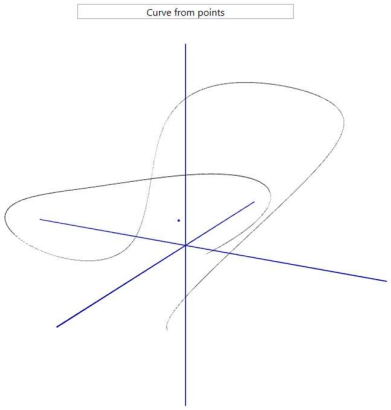




What if we have parametric curve given by a list of points?

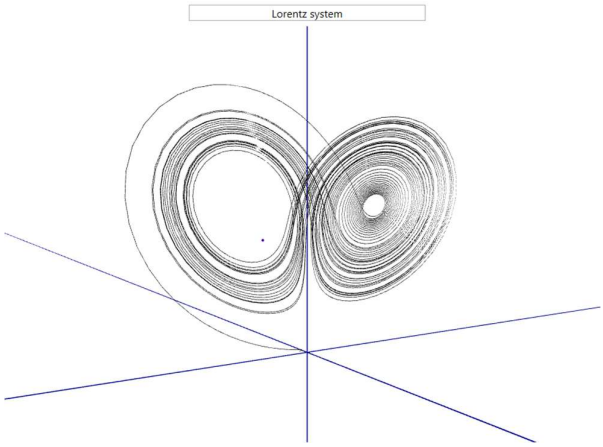
```
// First case - we already have points in Matrix
Matrix<Real> curve_points1{ 10, 3,
                           {0.0, 50.0, -100.0,
                            -100.0,-100.0, 50.0,
                            -150.0,-180.0, 150.0,
                            -80.0, -80.0, 200.0,
                            130.0,-150.0, 100.0,
                            150.0,-100.0, -50.0,
                            100.0, 180.0, 90.0,
                             50.0, 140.0, 100.0,
                            -70.0, 100.5, 120.0,
                             20.0, -100.5, -50.0} };

// we can easily create spline parametric curve
SplineInterpParametricCurve<3> curve(0.0, 1.0, curve_points1);

Visualizer::VisualizeParamCurve3D(curve, "Curve from points", 0.0, 1.0, 1000,
                                  "example3_wpf_curve_spline.txt");
```

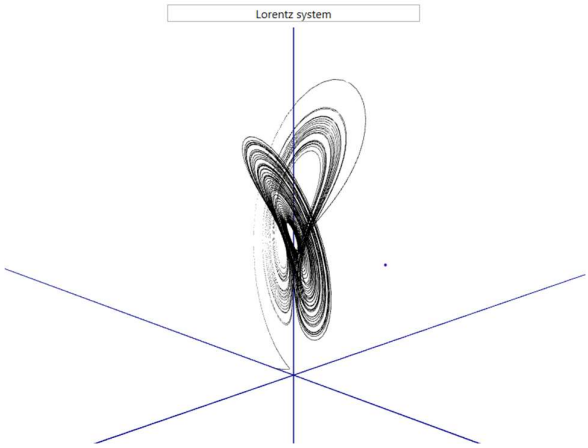


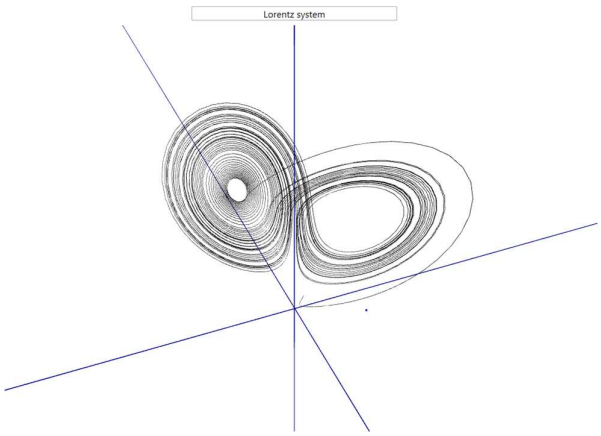


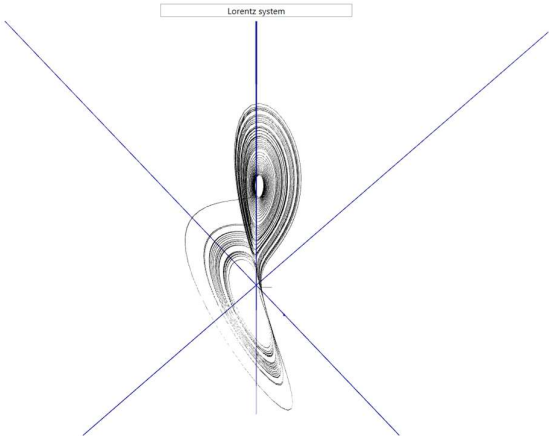
Visualizing Lorentz system solution as parametric curve in 3D.

```
// Lorentz system as 3D parametric curve
ODESystem lorentz_system(3, [](Real t, const Vector<Real>& x, Vector<Real>& dxdt)
  {
    const Real sigma = 10.0;
    const Real rho = 28.0;
    const Real beta = 8.0 / 3.0;
    dxdt[0] = sigma * (x[1] - x[0]);
    dxdt[1] = x[0] * (rho - x[2]) - x[1];
    dxdt[2] = x[0] * x[1] - beta * x[2];
  }
);

Vector<Real> ystart({ -2.0, 3.0, 1.0 });
ODESystemSolver<RK5_CashKarp_Stepper> solver(lorentz_system);

ODESystemSolution sol = solver.integrate(ystart, 0.0, 50.0, 0.001, 1e-08, 0.01);

SplineInterpParametricCurve<3> solCurve1 = sol.getSolutionAsParametricCurve3D(0, 1, 2);

Visualizer::VisualizeODESysSolAsParamCurve3(sol, 0, 1, 2, "Lorentz system",
                                            "example3_wpf_lorentz_system1.txt");
```



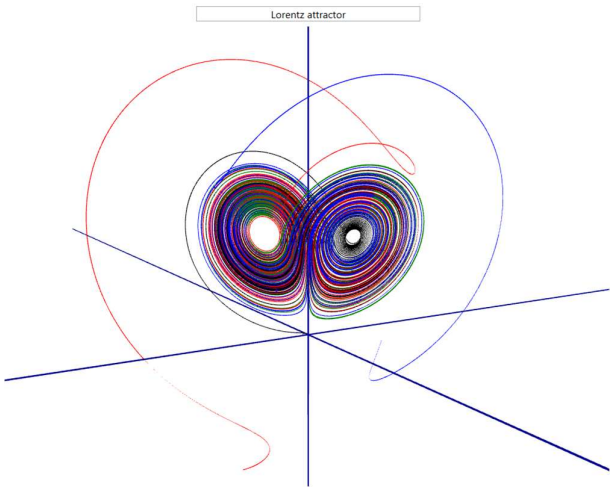


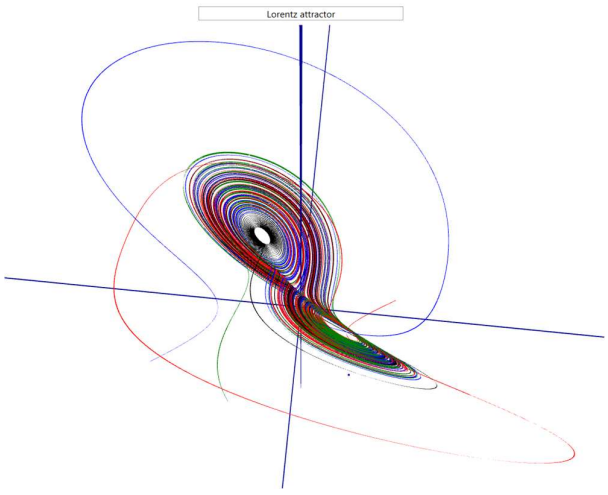


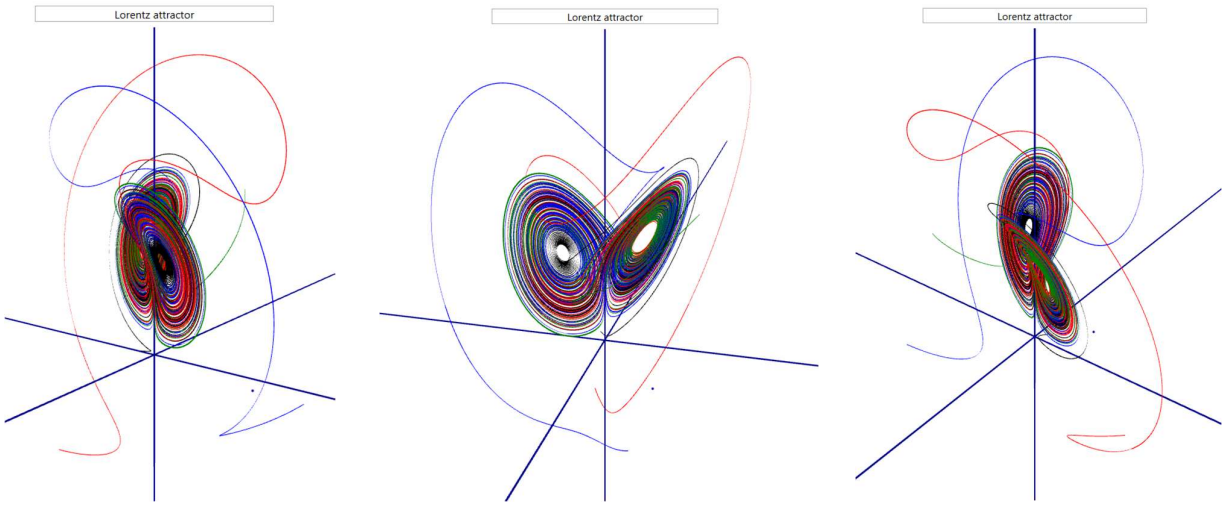


Adding more initial conditions.

```
Vector<Real> ystart1({5.0,-15.0,-10.0}), ystart2({-10.0,25.0,-3.0}), ystart3({15.0,-10.0,5.0});

sol = solver.integrate(ystart1, 0.0, 50.0, 0.001, 1e-10, 0.001);
SplineInterpParametricCurve<3> solCurve2 = sol.getSolutionAsParametricCurve3D(0, 1, 2);
sol = solver.integrate(ystart2, 0.0, 50.0, 0.001, 1e-10, 0.001);
SplineInterpParametricCurve<3> solCurve3 = sol.getSolutionAsParametricCurve3D(0, 1, 2);
sol = solver.integrate(ystart3, 0.0, 50.0, 0.001, 1e-10, 0.001);
SplineInterpParametricCurve<3> solCurve4 = sol.getSolutionAsParametricCurve3D(0, 1, 2);


// visualize together
Visualizer::VisualizeMultiParamCurve3D({ &solCurve1, &solCurve2 }, "Lorentz system",
                                       0.0, 50.0, 10000, "example3_wpf_lorentz_system_multi.txt");
```







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

Visualizing surfaces

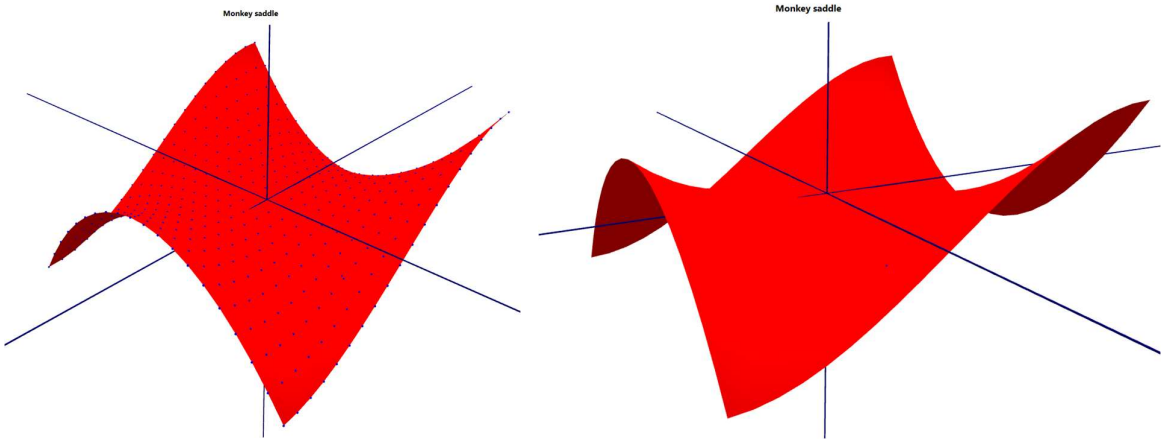
```
// Monkey saddle surface
ScalarFunction<2> testFunc1{ [](const VectorN<Real, 2>& x)
  {
    return 3 * x[0] / 5 * (x[0] * x[0] / 25 - 3 * x[1] * x[1] / 25);
  }
};

Visualizer::VisualizeScalarFunc2DCartesian(testFunc1, "Monkey saddle",
                                           -10.0, 10.0, 20, -10.0, 10.0, 20,
                                           "example3_wpf_surface1.txt");
```

Generated content in file:

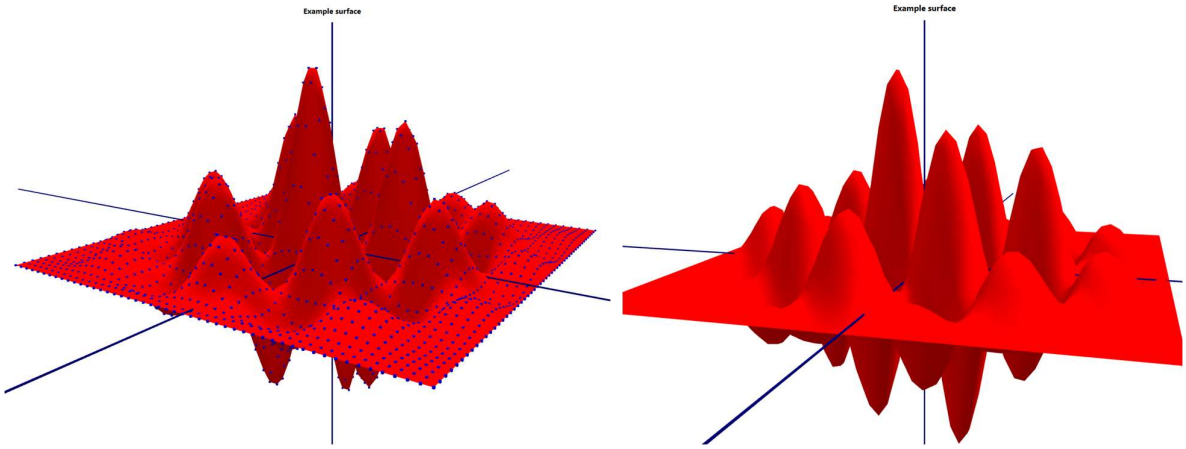
```
SCALAR_FUNCTION_CARTESIAN_2D
Monkey saddle
x1: -10
x2: 10
NumPointsX: 20
y1: -10
y2: 10
NumPointsY: 20
-10 -10 48
-10 -8.94737 33.6399
-10 -7.89474 20.8753
-10 -6.84211 9.70637
-10 -5.78947 0.132964
-10 -4.73684 -7.84488
-10 -3.68421 -14.2271
```

```
…
10 2.63158 19.0139
10 3.68421 14.2271
10 4.73684 7.84488
10 5.78947 -0.132964
10 6.84211 -9.70637
10 7.89474 -20.8753
10 8.94737 -33.6399
10 10 -48
```

Visualizations (with and without markings for surface points).



Second example.



Visualizing 2D & 3D vector fields

Visualizing vector field in 2 dimensions.

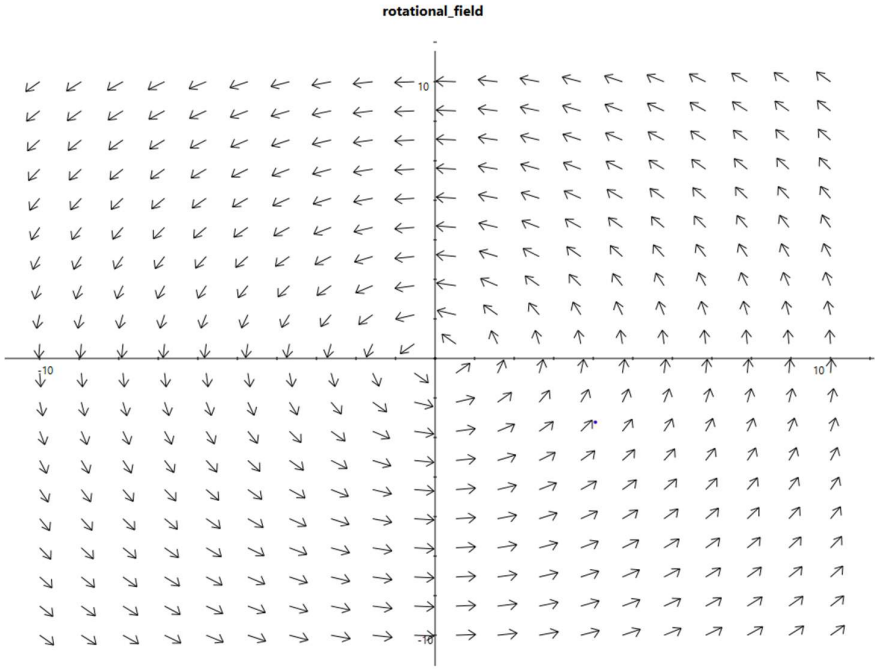
```
VectorFunction<2> rotational_field{ [](const VectorN<Real, 2>& x)
  {
    Real norm = x.NormL2();
    return VectorN<Real, 2>{-x[1] / norm, x[0] / norm};
  }
};

Visualizer::VisualizeVectorField2DCartesian(rotational_field, "rotational_field",
                                            -10.0, 10.0, 20, -10.0, 10.0, 20,
                                            "example3_wpf_vector_field_2d.txt");
```

Generated content in file.

```
VECTOR_FIELD_2D_CARTESIAN
rotational_field
-10 -10 0.707107 -0.707107
-10 -8.94737 0.666795 -0.745241
-10 -7.89474 0.619644 -0.784883
-10 -6.84211 0.564684 -0.825307
-10 -5.78947 0.501036 -0.865426
-10 -4.73684 0.428086 -0.903738
-10 -3.68421 0.345705 -0.938343
…
10 4.73684 -0.428086 0.903738
10 5.78947 -0.501036 0.865426
10 6.84211 -0.564684 0.825307
10 7.89474 -0.619644 0.784883
10 8.94737 -0.666795 0.745241
10 10 -0.707107 0.707107
```

Absence of x1, x2, NumPoints and similar parameters is due to the fact that visualizer component performs automatic analysis of input domain, so even though our serializer will save values in neat pattern, that is not a requirement.

Visualization.



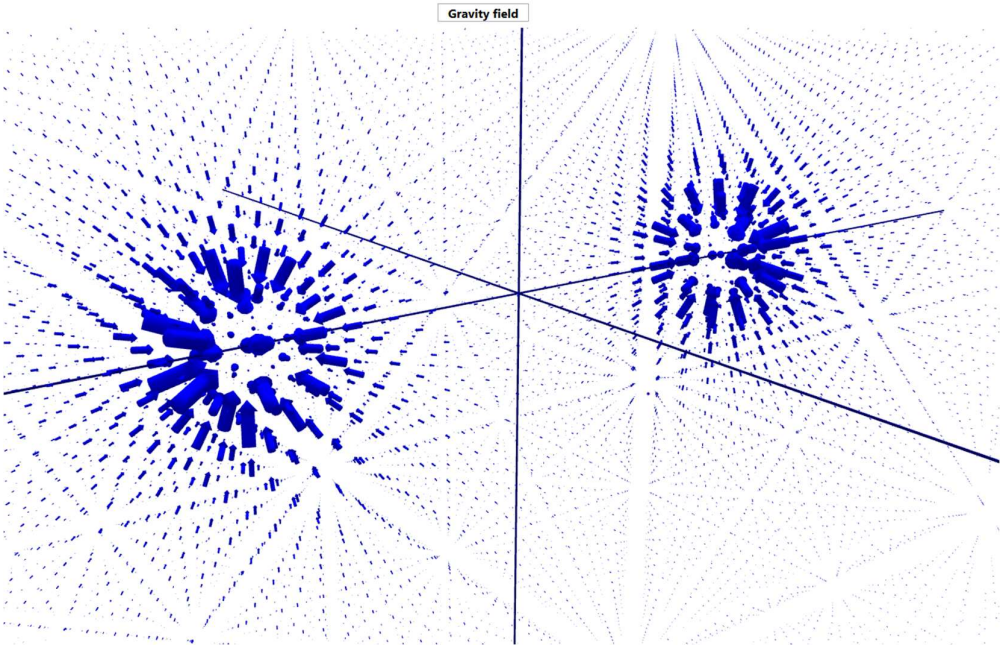
Visualizing vector fields in 3 dimensions

```
// defining force field of two masses as VectorFunction
VectorFunction<3> gravity_force_field{ [](const VectorN<Real, 3>& x)
{
    const VectorN<Real, 3> x1{ 100, 0, 0}, x2{ -100, 0, 0};
    const Real m1 = 100.0, m2 = 100.0;
    const Real G = 10;
    return -G * m1 * (x - x1) / std::pow((x - x1).NormL2(), 3) -
            G * m2 * (x - x2) / std::pow((x - x2).NormL2(), 3);
} };

Visualizer::VisualizeVectorField3DCartesian(gravity_force_field, "Gravity field",
                                            -200, 200, 30, -200, 200, 30, -200, 200, 30,
                                            "example3_wpf_vector_field_3d.txt");
```

Generated file content.

```
VECTOR_FIELD_3D_CARTESIAN
Gravity field
-200 -200 -200 0.00798374 0.0102608 0.0102608
-200 -200 -186.207 0.00854805 0.0111102 0.010344
-200 -200 -172.414 0.00914172 0.0120174 0.0103598
-200 -200 -158.621 0.00976156 0.0129785 0.0102933
-200 -200 -144.828 0.0104026 0.0139866 0.0101282
-200 -200 -131.034 0.0110579 0.015031 0.00984791
…
200 200 144.828 -0.0104026 -0.0139866 -0.0101282
200 200 158.621 -0.00976156 -0.0129785 -0.0102933
```

```
200 200 172.414 -0.00914172 -0.0120174 -0.0103598
200 200 186.207 -0.00854805 -0.0111102 -0.010344
200 200 200 -0.00798374 -0.0102608 -0.0102608
```



Visualizing particle simulations

Still a work in progress!

Used mostly for Collision simulator in Chapter 4, and N-body gravity simulation in Chapter 9

Enables visualization of N particles/bodies trajectories.

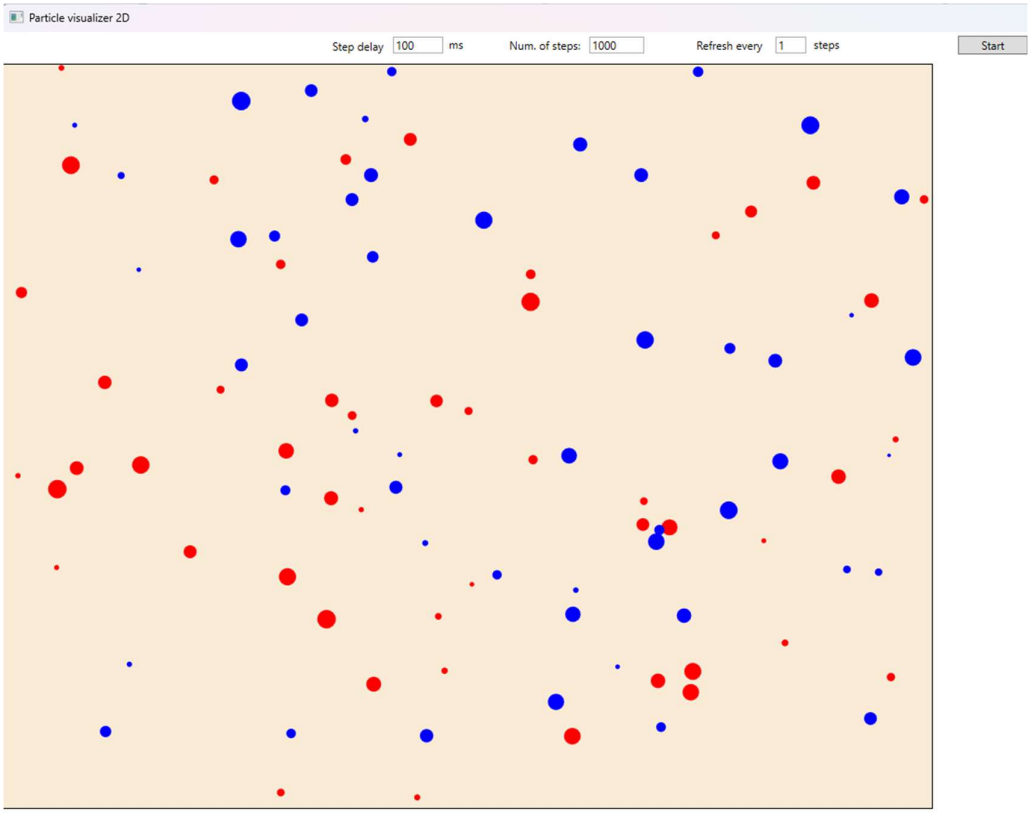
Particle simulation in 2D

Generated file with 100 balls (50 red, 50 blue), with random radiuses (in interval [3, 10], and random positions within container 1000 x 1000

File format:

```
PARTICLE_SIMULATION_DATA_2D
NumBalls: 100
Ball_1 Red 6.50868
Ball_2 Red 9.1677
Ball_3 Red 6.10828
Ball_4 Red 3.32719
Ball_5 Red 5.02304
Ball_6 Red 7.30436
…
```

```
Ball_95 Blue 5.1793
Ball_96 Blue 7.86956
Ball_97 Blue 4.18073
Ball_98 Blue 9.2595
Ball_99 Blue 2.72341
Ball_100 Blue 7.5783
…
NumSteps: 1000
Step 0 0.1
0 804.722 158.636
1 742.083 652.353
2 20.9098 245.59
3 446.054 787.569
…
97 517.628 167.955
98 78.0881 65.9395
99 396.442 119.555
Step 1 0.1
0 804.52 159.486
1 745.673 649.094
2 19.5543 241.504
3 441.1 783.76
…
96 373.656 241.988
97 731.294 513.615
98 491.409 292.996
99 872.334 719.965
```



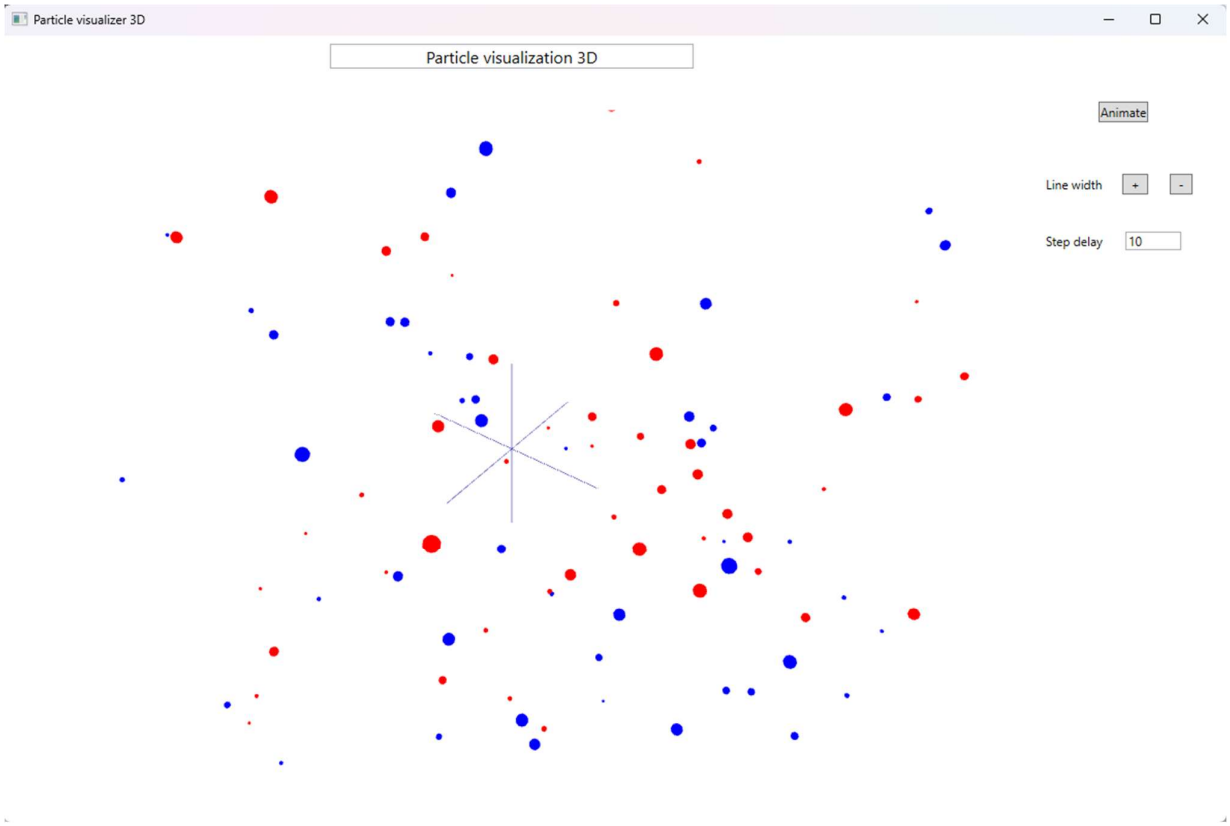
Particle simulation in 3D

Same as in previous example, but in 3D.

Generated file with trajectory data for 100 balls.

```
PARTICLE_SIMULATION_DATA_3D
NumBalls: 100
Ball_1 Red 8.2318
Ball_2 Red 8.1955
Ball_3 Red 3.45126
Ball_4 Red 7.48618
Ball_5 Red 3.66358
…
Ball_95 Blue 9.62492
Ball_96 Blue 8.90473
Ball_97 Blue 5.57198
Ball_98 Blue 9.37226
Ball_99 Blue 9.64663
Ball_100 Blue 6.69845
…
NumSteps: 100
Step 0 0.1
0 972.958 635.321 816.846
1 249.184 519.539 311.393
2 609.546 167.804 180.063
3 330.283 60.1874 701.593
4 115.636 292.918 219.628
5 181.193 119.617 137.269
…
96 269.386 601.865 499.579
97 567.563 741.41 153.786
98 576.764 161.905 137.72
99 158.822 830.989 838.669
Step 1 0.1
0 977.84 640.037 814.508
1 254.142 518.377 314.322
2 608.054 163.256 180.617
3 334.326 56.7461 701.798
…
95 46.8158 184.978 186.921
96 753.494 808.642 800.327
97 276.137 806.593 81.3732
98 802.489 195.024 298.528
99 74.7379 503.788 695.406
```

Visualization


Particle visualizer 3D
Particle visualization 3D
Animate
Line width
+
-
Step delay
10

3. Basic algorithms

Putting some algorithmic meat on our objects.

SOLVING SYSTEMS OF LINEAR EQUATIONS

General introduction

$$
\begin{aligned}
a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \cdots + a_{0,N-1}x_{N-1} &= b_0 \\
a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \cdots + a_{1,N-1}x_{N-1} &= b_1 \\
a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + \cdots + a_{2,N-1}x_{N-1} &= b_2 \\
\cdots & \cdots \\
a_{M-1,0}x_0 + a_{M-1,1}x_1 + \cdots + a_{M-1,N-1}x_{N-1} &= b_{M-1}
\end{aligned}
$$

TODO - 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> SolveConst(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

```
// initialize a 4x4 matrix, defining the system of linear equations
Matrix<Real> matOrig(4, 4, { -2,  4,  3,  5,
                              3,  8, -7, -2,
                              1, -3,  7,  4,
                              5, -4,  9,  1 });
Matrix<Real> mat1(matOrig), mat2(matOrig);
std::cout << "Initial matrix:\n";  matOrig.Print(std::cout, 10, 3);

// right side vector
Vector<Real> rhs({ 3, -2, 1, 5 }), sol1(rhs);
std::cout << "\nRight side: ";     rhs.Print(std::cout, 10, 3);

// solution is returned in given rhs vector (in place solution)
GaussJordanSolver<Real>::SolveInPlace(mat1, sol1);

// solution is returned in a new vector (rhs is not modified)
Vector<Real> sol2 = GaussJordanSolver<Real>::Solve(mat2, rhs);

// solution is returned in a new vector, with the original matrix unchanged
Vector<Real> sol3 = GaussJordanSolver<Real>::SolveConst(matOrig, rhs);

std::cout << "\nSol1 = " << sol1 << std::endl;
std::cout << "Sol2 = " << sol2 << std::endl;
std::cout << "Sol3 = " << sol3 << std::endl;

std::cout << "mat * sol1 = "; (matOrig * sol1).Print(std::cout, 10, 3);
std::cout << "\nmat * sol2 = "; (matOrig * sol2).Print(std::cout, 10, 3);
std::cout << "\nmat * sol3 = "; (matOrig * sol3).Print(std::cout, 10, 3);
```

Program output

```
Initial matrix:
Rows: 4 Cols: 4
[         -2,          4,          3,          5 ]
[          3,          8,         -7,         -2 ]
[          1,         -3,          7,          4 ]
[          5,         -4,          9,          1 ]
Right side: [          3,         -2,          1,          5]
Sol1 = [  -0.8427672956,     1.150943396,      1.723270044,    -1.691823899]
Sol2 = [  -0.8427672956,     1.150943396,      1.723270044,    -1.691823899]
Sol3 = [  -0.8427672956,     1.150943396,      1.723270044,    -1.691823899]
mat * sol1 = [          3,         -2,          1,          5]
mat * sol2 = [          3,         -2,          1,          5]
mat * sol3 = [          3,         -2,          1,          5]
```

TODO – solving complex system!

NUMERICAL DERIVATION

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

Mathematics of derivation approximation

Simple, right?

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Taylor expansion and all that jazz.

$$f(x+h) = f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + \tfrac{1}{6}h^3 f'''(x) + \cdots$$

Central difference is better, because it is a *second order* method

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

And it comes down to (mostly) properly handling value of ‘h’, depending on the order

Calculating second and third derivations

TODO

Derivation routines

Default step-sizes for routines of different order, 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);
```

As an example, here is 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;

    *error = ypph / 2 + (std::abs(yh) + std::abs(y0)) * Constants::Eps / h;
  }
  return diff / h;
}
```

Second order method, using central difference.

```
static Real NDer2(const IRealFunction& f, Real x, Real h, Real* error = nullptr)
{
  Real yh = f(x + h);
  Real ymh = f(x - h);
  Real diff = yh - ymh;
  if (error)
  {
    Real y2h = f(x + 2 * h);
    Real ym2h = f(x - 2 * h);
    *error = Constants::Eps * (std::abs(yh) + std::abs(ymh)) / (2 * h) +
             std::abs((y2h - ym2h) / 2 - diff) / (6 * h);
  }
  return diff / (2 * h);
}
```

Calculating fourth order derivation requires four function evaluations.

```
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);
}
```

And a similar example for deriving VectorFunction (of second order, for simplicity), where we have to specify both function index, and index of variable we want to derive by).

```
template <int N>
static Real NDer2Partial(const IVectorFunction<N>& f, int func_index, 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)[func_index];

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

  Real diff = yh - ymh;

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

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

    *error = Constants::Epsilon * (std::abs(yh) +  std::abs(ymh)) / (2 * h) +
             std::abs((y2h - ym2h) / 2 - diff) / (6 * h);
  }

  return diff / (2 * h);
}
```

RealFunction derivation routines

Given are definitions only for first order, but, as with all other types, all presented functions are available up to 8th order.

```
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) { ... }
```

For RealFunction we also have second and third derivations

```
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) { ... }
```

ScalarFunction derivation routines

```
template <int N>
static Real NDer1Partial(const IScalarFunction<N>& f, int deriv_index, const VectorN<Real, N>& point,
                         Real h, Real* error = nullptr) { ... }
template <int N>
static Real NDer1Partial(const IScalarFunction<N>& f, int deriv_index, const VectorN<Real, N>& point,
                         Real* error = nullptr) { ... }
template <int N>
static VectorN<Real, N> NDer1PartialByAll(const IScalarFunction<N>& f, const VectorN<Real, N>& point,
                                          Real h, VectorN<Real, N>* error = nullptr) { ... }
template <int N>
static VectorN<Real, N> NDer1PartialByAll(const IScalarFunction<N>& f, const VectorN<Real, N>& point,
                                          VectorN<Real, N>* error = nullptr) { ... }
```

We can also calculate second partial derivations for ScalarFunction

```
template <int N>
static Real NSecDer1Partial(const IScalarFunction<N>& f, int der_ind1, int der_ind2,
                            const VectorN<Real, N>& point, Real h, Real* error = nullptr) { ... }
template <int N>
static Real NSecDer1Partial(const IScalarFunction<N>& f, int der_ind1, int der_ind2,
                            const VectorN<Real, N>& point, Real* error = nullptr) { ... }
```

VectorFunction derivation routines

```
template <int N>
static Real NDer1Partial(const IVectorFunction<N>& f, int func_index, int deriv_index,
                         const VectorN<Real, N>& point, Real h, Real* error = nullptr) { ... }

template <int N>
static Real NDer1Partial(const IVectorFunction<N>& f, int func_index, int deriv_index,
                         const VectorN<Real, N>& point, Real* error = nullptr) { ... }

template <int N>
static VectorN<Real, N> NDer1PartialByAll(const IVectorFunction<N>& f, int func_index,
                                          const VectorN<Real, N>& point, Real h, VectorN<Real, N>* error = nullptr)

template <int N>
static VectorN<Real, N> NDer1PartialByAll(const IVectorFunction<N>& f, int func_index,
                                          const VectorN<Real, N>& point, VectorN<Real, N>* error = nullptr) { ... }

template <int N>
static MatrixNM<Real, N, N> NDer1PartialAllByAll(const IVectorFunction<N>& f, const VectorN<Real, N>& point,
                                                 Real h, MatrixNM<Real, N, N>* error = nullptr) { ... }

template <int N>
static MatrixNM<Real, N, N> NDer1PartialAllByAll(const IVectorFunction<N>& f, const VectorN<Real, N>& point,
                                                 MatrixNM<Real, N, N>* error = nullptr) { ... }
```

ParametricCurve derivation routines

```
template <int N>
static VectorN<Real, N> NDer1(const IParametricCurve<N>& f, Real t, Real h, Real* error = nullptr) { ... }

template <int N>
static VectorN<Real, N> NDer1(const IParametricCurve<N>& f, Real t, Real* error = nullptr) { ... }
```

Second and third derivations are also available for ParametricCurve

```
template <int N>
static VectorN<Real, N> NSecDer1(const IParametricCurve<N>& f, Real x, Real h, Real* error = nullptr) { ... }
template <int N>
static VectorN<Real, N> NSecDer1(const IParametricCurve<N>& f, Real x, Real* error = nullptr) { ... }

template <int N>
static VectorN<Real, N> NThirdDer1(const IParametricCurve<N>& f, Real x, Real h, Real* error = nullptr) { ... }
template <int N>
static VectorN<Real, N> NThirdDer1(const IParametricCurve<N>& f, Real x, Real* error = nullptr) { ... }
```

Examples of usage - derivation

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

RealFunction derivation examples

```
RealFunction       f1{ [](Real x) { return (Real)(sin(x) * (1.0 + 0.5 * x * x)); } };

// numerical derivation of real function (available orders - 1, 2, 4, 6, 8)
double der_f1 = Derivation::NDer1(f1, 0.5);
double der_f2 = Derivation::NDer2(f1, 0.5, 1e-6);   // setting explicit step size
Real err;
double der_f6 = Derivation::NDer6(f1, 0.5, &err);   // if we need error estimate
double der_f8 = Derivation::NDer8(f1, 0.5);

// we can use default Derive routine (set to NDer4)
double der_f4 = Derivation::Derive(f1, 0.5);
// available also with error estimate
double der_f41 = Derivation::DeriveErr(f1, 0.5, &err);

// second and third derivatives
double sec_der_f1   = Derivation::NSecDer2(f1, 0.5);
double third_der_f1 = Derivation::NThirdDer2(f1, 0.5);
```

Scalar and vector functions

```
ScalarFunction<3>   f2Scal([](const VectorN<Real, 3>& x) { return (Real)(1.0 / pow(x.NormL2(), 2)); });
VectorFunction<3>   f3Vec([](const VectorN<Real, 3>& x) { return VectorN<Real, 3>{0, x[0] * x[1], 0}; });

VectorN<Real, 3> der_point{ 1.0, 1.0, 1.0 };
```

```
double           der_f2Scal     = Derivation::NDer1Partial(f2Scal, 1, der_point);
VectorN<Real, 3> der_f2Scal_all = Derivation::NDer1PartialByAll(f2Scal, der_point);

double               der_f3Vec        = Derivation::NDer1Partial(f3Vec, 1, 1, der_point);
VectorN<Real, 3>     der_f3Vec_by1    = Derivation::NDer2PartialByAll(f3Vec, 1, der_point);
MatrixNM<Real, 3, 3> der_f3Vec_by_all = Derivation::NDer4PartialAllByAll(f3Vec, der_point);
```

Parametric curve

```
ParametricCurve<3>  f4Curve([](Real x) { return VectorN<Real, 3>{x, 2 * x, 3 * x}; });

VectorN<Real, 3>    der_f4Curve   = Derivation::NDer1(f4Curve, 1.0);
VectorN<Real, 3>    sec_f4Curve   = Derivation::NSecDer1(f4Curve, 1.0);
VectorN<Real, 3>    third_f4Curve = Derivation::NThirdDer1(f4Curve, 1.0);
```

Automatic differentiation

Short overview of basic AD implementation

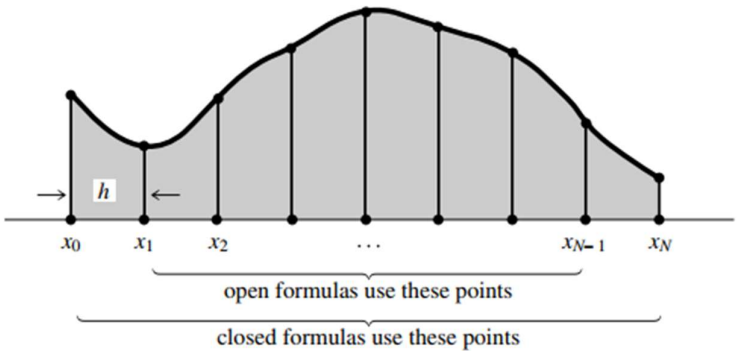
NUMERICAL INTEGRATION

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.

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

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 of trapezoidal integration function

```
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

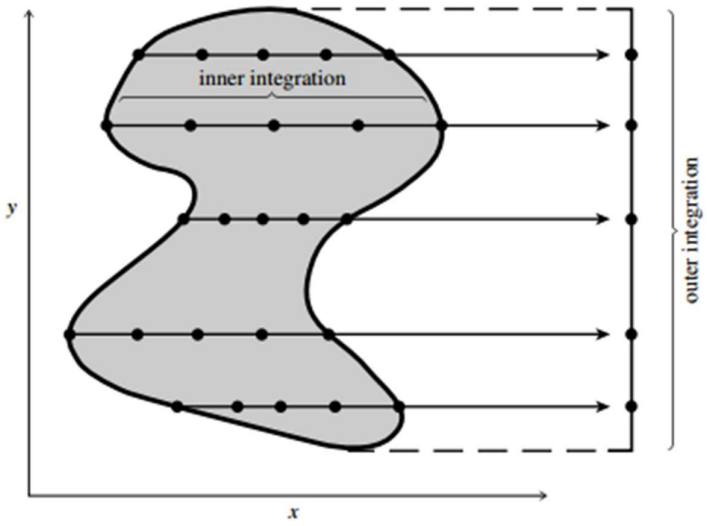
https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_integration.cpp

```
RealFunction f1{ [](Real x) { return (Real)(sin(x) * (1.0 + 0.5 * x * x)); } };
RealFunction f1_integral{ [](Real x) { return (Real)(x * (-0.5 * x * cos(x) + sin(x))); } };

double a = 0.0;
double b = 10.0;
double int_trap  = IntegrateTrap(f1, a, b);
double int_simp  = IntegrateSimpson(f1, a, b);
double int_gauss = IntegrateGauss10(f1, a, b);

std::cout << "Integrating function f1 from " << a << " to " << b << std::endl;
std::cout << "Exact integral   = " << f1_integral(b) - f1_integral(a) << std::endl;
std::cout << "IntegrateTrap    = " << int_trap << std::endl;
std::cout << "IntegrateSimpson = " << int_simp << std::endl;
std::cout << "IntegrateGauss10 = " << int_gauss << std::endl;

/* OUTPUT
Integrating function f1 from 0 to 10
Exact integral   = 36.51336534
IntegrateTrap    = 36.51297408
IntegrateSimpson = 36.51337665
IntegrateGauss10 = 36.51336529
*/
```

Integrating in 2D

TODO – intro

This is what it means to integrate numerically 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

```
// 2D integration of constant 2D function over area (ie. we'll get the area of the surface)
ScalarFunction<2> f2([](const VectorN<Real, 2>& x) { return Real{ 1 }; });

// we integrate over circle with radius 2
Real  val = Integrate2D(f2, IntegrationMethod::GAUSS10,
                        -2, 2,              // x range
                        [](Real x) { return -sqrt(4 - x * x); },   // y range lower limit
                        [](Real x) { return sqrt(4 - x * x); });   // y range upper limit

std::cout << "Circle area = " << val << ", exact value: 4 * PI = " << 4 * Constants::PI << std::endl;

// calculate volume of half-sphere
ScalarFunction<2> f3([](const VectorN<Real, 2>& x) { return Real{ sqrt(4 - POW2(x[0]) - POW2(x[1])) }; });
Real val2 = Integrate2D(f3, IntegrationMethod::GAUSS10,
                        -2, 2,              // x range
                        [](Real x) { return -sqrt(4 - x * x); },   // y range lower limit
                        [](Real x) { return sqrt(4 - x * x); });   // y range upper limit

std::cout << "Half-sphere volume  = " << val2 << ", exact value = " << 4.0 / 6 * 8 * Constants::PI << std::endl;

/* OUTPUT
    Calc. area = 12.57211164, exact value: 4 * PI = 12.56637061
    Calc. half-sphere volume  = 16.76281553, exact value = 16.75516082
*/
```

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

```cpp
// 3D integration of constant scalar 3D function (ie. we'll get the volume of the solid)
ScalarFunction<3> f3([](const VectorN<Real, 3>& x) { return Real{ 1 }; });

// integration over cube with sides 2, 3, 4
Real  cubeVol = Integrate3D(f3,
                            -1, 1,
                            [](Real x) { return -1.5; },
                            [](Real x) { return 1.5; },
                            [](Real x, Real y) { return -2.0; },
                            [](Real x, Real y) { return 2.0; });

std::cout << "Cube vol. = " << cubeVol << ", exact value: 2 * 3 * 4 = " << 2 * 3 * 4 << std::endl;

// integration over sphere of radius 1
Real  sphereVol = Integrate3D(f3,
                              -1, 1,
                              [](Real x) { return -sqrt(1 - x * x); },
                              [](Real x) { return sqrt(1 - x * x); },
                              [](Real x, Real y) { return -sqrt(1 - x * x - y * y); },
                              [](Real x, Real y) { return sqrt(1 - x * x - y * y); });

std::cout << "Sphere vol. = " << sphereVol << ", exact value: 4/3*PI = " << 4.0/3*Constants::PI << std::endl;

/* OUTPUT
    Cube vol. = 24, exact value: 2 * 3 * 4 = 24
    Sphere vol. = 4.1907, exact value: 4/3 * PI = 4.18879
*/
```

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

Using SplineInterpFunc objects, constructs cubic spline interpolation for parametric curve in N dimension.

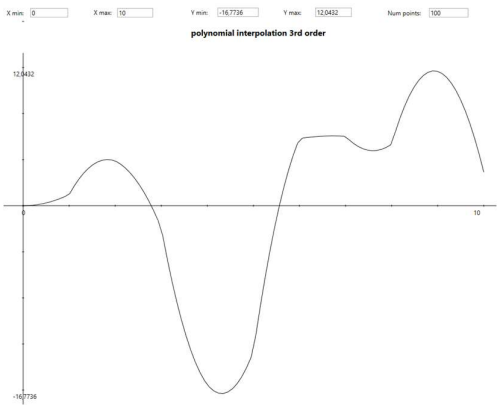
```
// Object for interpolating a curve specified by _numPoints points in N dimensions.
template<int N>
class SplineInterpParametricCurve : public IParametricCurve<N>
{
  Real _minT, _maxT;
  int  _dim, _numPoints, _bemba;
  bool _isCurveClosed;

  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) { ... }
};
```

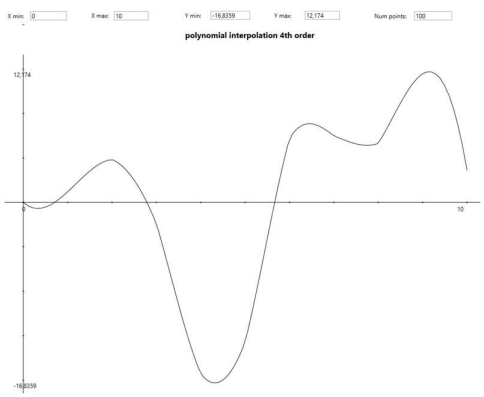
Example usage – interpolating functions

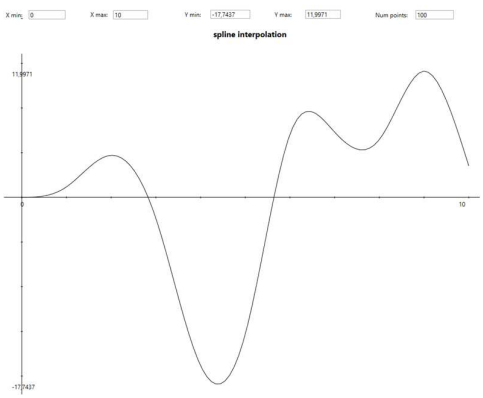
https://github.com/zvanjak/ExploringPhysicsWithCpp/blob/master/Code/Chapter_02_basic_algorithms/demo_interpolation.cpp

```
// we are manually creating data for interpolation in two vectors
// but generally you would have some data from some source
Real x1 = 0.0;
Real x2 = 10.0;
Vector<Real> vec_x{ 0.0, 1.0, 2.0,  3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 };
Vector<Real> vec_y{ 0.0, 1.0, 4.0, -2.0, -16, -13,   6, 6.2, 5.5, 12.0, 3.0 };

// create InterpolatedFunction object
LinearInterpRealFunc    lin_interp(vec_x, vec_y);
PolynomInterpRealFunc   poly_interp3(vec_x, vec_y, 3);
PolynomInterpRealFunc   poly_interp4(vec_x, vec_y, 4);
SplineInterpRealFunc    spline_interp(vec_x, vec_y);

// visualize interpolations
Visualizer::VisualizeRealFunction(lin_interp, "linear interpolation",
                                  x1, x2, 100, "example2_lin_interp.txt");
Visualizer::VisualizeRealFunction(poly_interp3, "polynomial interpolation 3rd order",
                                  x1, x2, 100, "example2_poly_interp3.txt");
Visualizer::VisualizeRealFunction(poly_interp4, "polynomial interpolation 4th order",
                                  x1, x2, 100, "example2_poly_interp4.txt");
Visualizer::VisualizeRealFunction(spline_interp, "spline interpolation",
                                  x1, x2, 100, "example2_spline_interp.txt");

// visualize all together
Visualizer::VisualizeMultiRealFunction({&lin_interp, &poly_interp3, &poly_interp4, &spline_interp},
                                       "Interpolation comparison",
                                       0.0, 10.0, 100, "example2_interpolation.txt" );
```

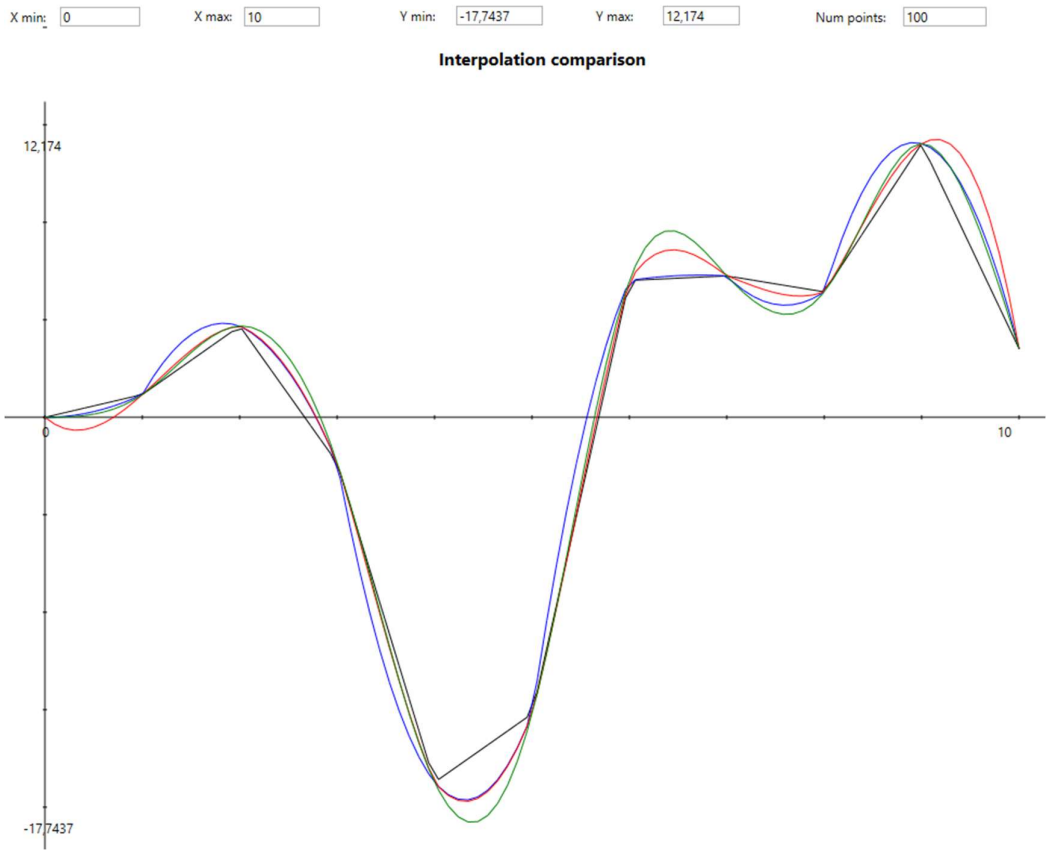
Visualizations







All together.



ROOT FINDING

Analytically finding roots of functions is possible in a very limited set of cases.

Polynomic roots up to 4th order

We can use functions from MMLBase.h

```
// 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) { ... }
```

And we mostly have to rely on numerical methods.

Bracketing roots

Two functions that help with 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

Simple example of finding (single) root of a function.

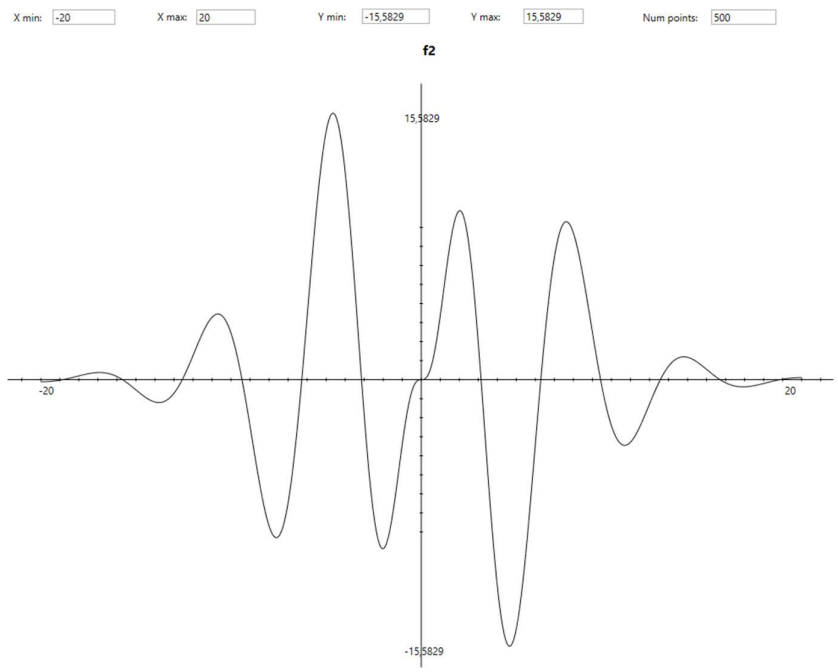
```
// Finding the root of the function f(x) = x^2 - 2
// (with known root sqrt(2) = 1.414213562373095)
RealFunction ff{ [](Real x) { return x * x - 2; } };
Real a = 0.0;
Real b = 10.0;

Real root = FindRootBisection(f, a, b, 0.0001);

std::cout << "Root of the function using the bisection method: " << root << std::endl;
std::cout << "Exact root: " << sqrt(2) << std::endl;
std::cout << "Error: " << abs(root - sqrt(2)) << std::endl;
```

Example with function with multiple roots.

```
Real x1 = -20.0;
Real x2 = 20.0;
RealFunction f2{ [](Real x) { return x * x * sin(x) * exp(2-Abs(x/2)); } };

int   numFoundRoots;
Vector<Real> root_brack_x1(10), root_brack_x2(10);
FindRootBrackets(f2, x1, x2, 100, root_brack_x1, root_brack_x2, numFoundRoots);

std::cout << "Number of found roots: " << numFoundRoots << std::endl;

Vector<Real> roots(numFoundRoots);
for (int i = 0; i < numFoundRoots; i++)
{
  roots[i] = FindRootBisection(f2, root_brack_x1[i], root_brack_x2[i], 1e-7);

  std::cout << "Root " << i << " : " << roots[i] << std::endl;
}

// visualize the function
Visualizer::VisualizeRealFunction(f2, "f2", x1, x2, 500, "example2_root_finding.txt");
```

Program output and function visualization.

```
Number of found roots: 13
Root 0 : -18.8496
Root 1 : -15.708
Root 2 : -12.5664
Root 3 : -9.42478
Root 4 : -6.28319
Root 5 : -3.14159
Root 6 : -4.10783e-15
Root 7 : 3.14159
Root 8 : 6.28319
Root 9 : 9.42478
Root 10 : 12.5664
Root 11 : 15.708
Root 12 : 18.8496
```



NUMERICALLY SOLVING (SYSTEMS OF) DIFFERENTIAL EQUATIONS

Mathematical introduction

Importance of initial conditions

- Initial-value problems
- Boundary-value problems

When numerically solving differential equations, it is always possible to reduce them to a set of coupled *first-order* ODE's. For example, the second-order equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

Can be reduced to a system of two first-order equations

$$\frac{dy}{dx} = z(x)$$
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

With introduction of new variable z(x).

Therefore, all numerical solvers are aimed at solving a set of N first-order equations for functions y_i, i = 0, .., N-1, with the general form:

$$\frac{dy_i(x)}{dx} = f_i(x, y_0, \ldots, y_{N-1}), \qquad i = 0, \ldots, N-1$$

Euler method

Formula for Euler method:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

Midpoint method

Formula:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right)$$
$$y_{n+1} = y_n + k_2 + O(h^3)$$

Runge-Kutta methods

Formula:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1)$$
$$k_3 = hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2)$$
$$k_4 = hf(x_n + h, y_n + k_3)$$
$$y_{n+1} = y_n + \tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4 + O(h^5)$$

Adaptive step size methods

Richardson extrapolation (Bulirsch-Stoer)

Predictor-corrector multi-step methods

IMPLEMENTING ODE SOLVERS IN C++

First question is – how to model system of ODE's?

One approach is with templates, relying on overload of operator() to provide calculation of derivatives.

We will take another approach, using interfaces to directly model the ODE system.

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;

  int  _sys_dim;
  int  _totalSavedSteps;

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

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

Public functions.

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

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

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

bool Serialize(std::string fileName, std::string title) const { ... }
bool SerializeAsParametricCurve3D(std::string fileName, std::string title) const { ... }
```

Step calculators

First and basic ingredient in any numerical ODE solver.

Have general interface defined in IODESystemStepCalculators.h

```
class IODESystemStepCalculator
{
public:
  virtual void calcStep(const IODESystem& odeSystem,
                        const Real t, const Vector<Real>& x_start, const Vector<Real>& dxdt,
                        const Real h, Vector<Real>& xout, Vector<Real>& xerr) const = 0;
};
```

Short description of all step calculators is that, for a given IODESystem of dimension n, and given initial values for the variables x_start[0..n-1] and their derivatives dxdt[0..n-1] known at t, they used specific method to advance the solution over an interval h and return the incremented variables as x_out[0..n-1].

Euler method

```
class EulerStep_Calculator : public IODESystemStepCalculator
{
public:
  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();

    for (i = 0; i < n; i++)
      x_out[i] = x_start[i] + h * dxdt[i];
  }
};
```

Euler-Cromer method

With small modification, this step calculator, unlike original Euler method, preserves phase space

```
class EulerCromer_StepCalculator : public IODESystemStepCalculator
{
public:
  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> x_new(n);

    for (i = 0; i < n; i++)
      x_new[i] = x_start[i] + h * dxdt[i];

    odeSystem.derivs(t + h, x_new, x_out);
  }
};
```

Midpoint method

```
class MidpointStep_Calculator : public IODESystemStepCalculator
{
public:
  // ...
  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> x_mid(n);

    for (i = 0; i < n; i++)
      x_mid[i] = x_start[i] + 0.5 * h * dxdt[i];

    odeSystem.derivs(t + 0.5 * h, x_mid, x_out);
  }
};
```

Basic Runge-Kutta 4th order

```
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]);
  }
};
```

Basic Runge-Kutta 5th order, with embedded error calculation

As the simplest example of all modern ODE solvers, uses 5th order Runge-Kutta step to advance solution, and embedded (meaning, no new calls of derivs() function needed!) 4th order method to estimate error.

```
class RK5_CashKarp_Calculator : public IODESystemStepCalculator
{
public:

  void calcStep(const IODESystem& sys,
                Real t, const Vector<Real>& x, const Vector<Real>& dxdt,
                Real h, Vector<Real>& x_out, Vector<Real>& x_err) const override
  {
    static const Real a2 = 0.2, a3 = 0.3, a4 = 0.6, a5 = 1.0, a6 = 0.875,
      b21 = 0.2, b31 = 3.0 / 40.0, b32 = 9.0 / 40.0, b41 = 0.3, b42 = -0.9,
      b43 = 1.2, b51 = -11.0 / 54.0, b52 = 2.5, b53 = -70.0 / 27.0,
      b54 = 35.0 / 27.0, b61 = 1631.0 / 55296.0, b62 = 175.0 / 512.0,
      b63 = 575.0 / 13824.0, b64 = 44275.0 / 110592.0, b65 = 253.0 / 4096.0,
      c1 = 37.0 / 378.0, c3 = 250.0 / 621.0, c4 = 125.0 / 594.0, c6 = 512.0 / 1771.0,
      dc1 = c1 - 2825.0 / 27648.0, dc3 = c3 - 18575.0 / 48384.0,
      dc4 = c4 - 13525.0 / 55296.0, dc5 = -277.00 / 14336.0, dc6 = c6 - 0.25;

    int i, n = x.size();
    Vector<Real> ak2(n), ak3(n), ak4(n), ak5(n), ak6(n), xtemp(n);

    for (i = 0; i < n; i++)
      xtemp[i] = x[i] + b21 * h * dxdt[i];

    sys.derivs(t + a2 * h, xtemp, ak2);
    for (i = 0; i < n; i++)
      xtemp[i] = x[i] + h * (b31 * dxdt[i] + b32 * ak2[i]);

    sys.derivs(t + a3 * h, xtemp, ak3);
    for (i = 0; i < n; i++)
      xtemp[i] = x[i] + h * (b41 * dxdt[i] + b42 * ak2[i] + b43 * ak3[i]);

    sys.derivs(t + a4 * h, xtemp, ak4);
    for (i = 0; i < n; i++)
      xtemp[i] = x[i] + h * (b51 * dxdt[i] + b52 * ak2[i] + b53 * ak3[i] + b54 * ak4[i]);

    sys.derivs(t + a5 * h, xtemp, ak5);
    for (i = 0; i < n; i++)
      xtemp[i] = x[i] + h * (b61 * dxdt[i] + b62 * ak2[i] + b63 * ak3[i] + b64 * ak4[i] + b65 * ak5[i]);

    sys.derivs(t + a6 * h, xtemp, ak6);

    for (i = 0; i < n; i++)
      x_out[i] = x[i] + h * (c1 * dxdt[i] + c3 * ak3[i] + c4 * ak4[i] + c6 * ak6[i]);
    for (i = 0; i < n; i++)
      x_err[i] = h * (dc1 * dxdt[i] + dc3 * ak3[i] + dc4 * ak4[i] + dc5 * ak5[i] + dc6 * ak6[i]);
  }
};
```

Fixed-step ODE solver

For a given ODESystem and StepCalculator, performs integration of ODESystem with a given number of fixed-size steps. Can be used with any StepCalculator.

```
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
{
protected:
  // references that stepper gets from the solver
  const IODESystem&   _sys;

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

  Real  _absTol, _relTol;
  Real  _eps;

  Real  _tOld;
  Real  _hDone, _hNext;

  Vector<Real> _xout, _xerr;
```

Its interface

```
public:
  StepperBase(const 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;
```

Runge-Kutta 5th order Cash-Karp adaptive size algorithm

Example of simple adaptive size algorithm

5th order algorithm that uses embedded 4th order method to evaluate accuracy

```
class RK5_CashKarp_Stepper : public StepperBase
{
private:
  RK5_CashKarp_Calculator _stepCalc;

public:
  // Initializing stepper with references to the solver's system, time, state and derivative vectors.
  RK5_CashKarp_Stepper(const IODESystem& sys, Real& t, Vector<Real>& x, Vector<Real>& dxdt)
    : StepperBase(sys, t, x, dxdt) { }

  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] = std::abs(_x[i]) + std::abs(_dxdt[i] * h) + 1e-25;

    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 ODESolverError("Stepsize underflow in RK5_CashKarp_Stepper::doStep()");
    }
    // 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;
    _x = xtemp;
  }
};
```

Dormand-Prince 5th order

Dormand-Prince 8th order as top dog

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");
}
```

4. From colliding mechanical balls to (ideal) gas simulation

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 evolution of their positions in time.
- Calculate pressure of the balls on container walls
- Relate to gas laws

PHYSICS OF COLLIDING MECHANICAL BALLS

Starting with basics of physics

We'll need First and Third Newton law (but not the Second!), 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:

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

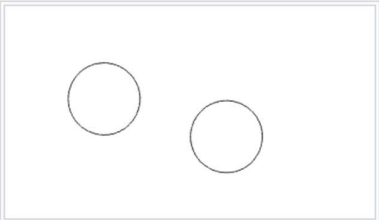
When solved, gives:

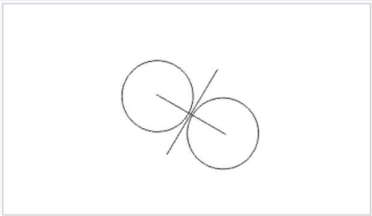
$$\begin{aligned} v_{A2} &= \frac{m_A - m_B}{m_A + m_B} v_{A1} + \frac{2m_B}{m_A + m_B} v_{B1} \\ v_{B2} &= \frac{2m_A}{m_A + m_B} v_{A1} + \frac{m_B - m_A}{m_A + m_B} v_{B1}. \end{aligned}$$

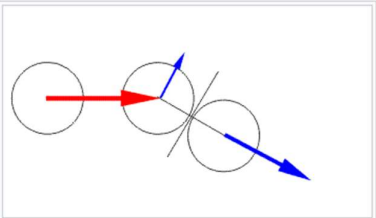
Center of mass frame doesn't change!

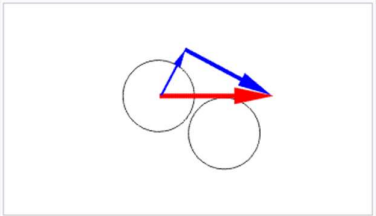
Two-dimensional case

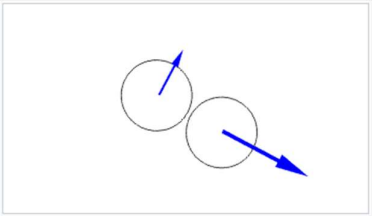
Visualization from Wikipedia:

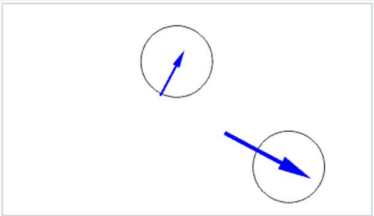

Two-dimensional elastic collision


Two-dimensional elastic collision


Two-dimensional elastic collision


Two-dimensional elastic collision


Two-dimensional elastic collision


Two-dimensional elastic collision

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:

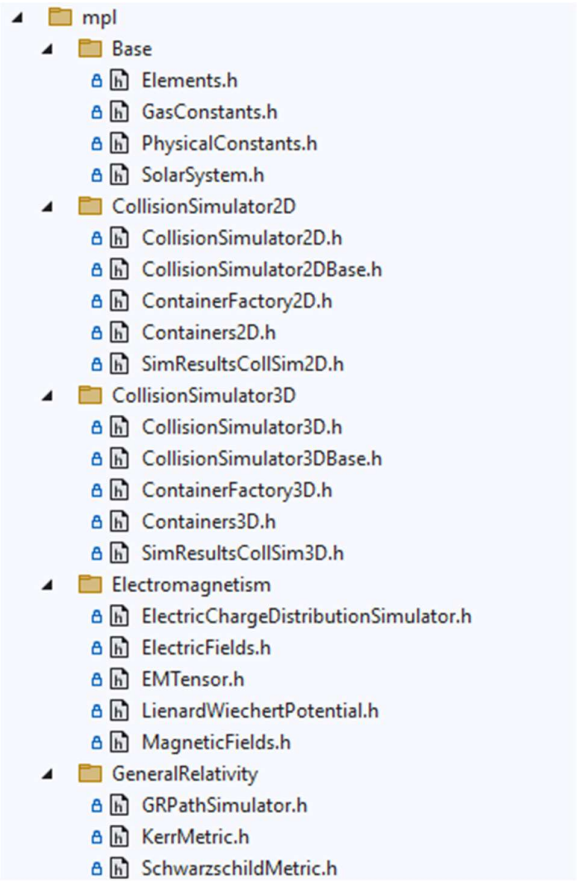
$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1+m_2}\frac{\langle \mathbf{v}_1-\mathbf{v}_2, \mathbf{x}_1-\mathbf{x}_2\rangle}{\|\mathbf{x}_1-\mathbf{x}_2\|^2}(\mathbf{x}_1-\mathbf{x}_2),$$
$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1+m_2}\frac{\langle \mathbf{v}_2-\mathbf{v}_1, \mathbf{x}_2-\mathbf{x}_1\rangle}{\|\mathbf{x}_2-\mathbf{x}_1\|^2}(\mathbf{x}_2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

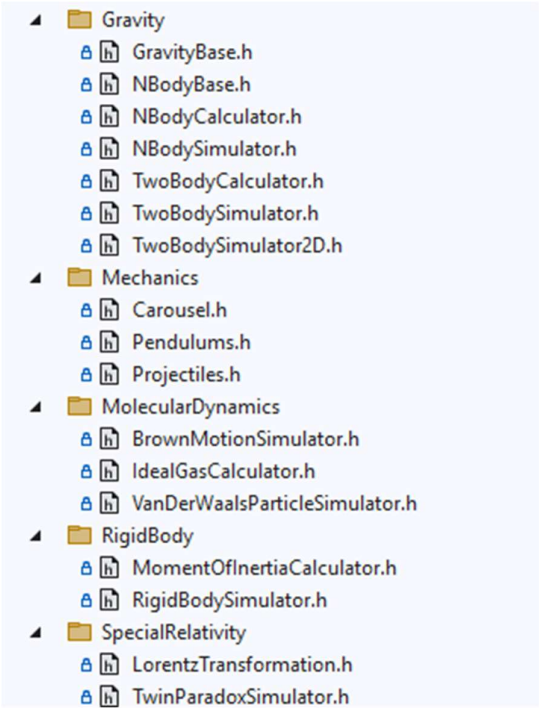
Based on these equations, we can easily implement calculations of velocity vectors for two balls after collision (but, notice, it depends on positions and velocities at the exact moment of collision!).

```
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);
```

STARTING OUR MINIMAL PHYSICS LIBRARY

In the first three chapters we have given overview of our Minimal Math Library, focused on providing basic (numerical) mathematics algorithms, and as we now move onto proper domain of physics, it is time for another library.





This library will contain all our code relevant to physics, with two focuses:

1) Modelling physical objects (using MML)
2) Implementing calculations/simulations of physical scenarios (again, using MML)

BUILDING COLLISIONSIMULATOR2D

Our goal given at the start of the Chapter - given positions and initial velocities of N simple mechanical balls, confined to 2D or 3D container, calculate evolution of their positions in time.

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, _radius;
  std::string _color;
  Pnt2Cart _position;
  Vec2Cart _velocity;
```

And its public interface, enabling easy manipulation:

```
public:
  Ball2D(double mass, double radius, std::string color, const Pnt2Cart& position, const Vec2Cart& velocity)
    : _mass(mass), _radius(radius), _color(color), _position(position), _velocity(velocity) { }

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

  Pnt2Cart  Pos() const { return _position; }
  Pnt2Cart& Pos()       { return _position; }
  Vec2Cart  V()   const { return _velocity; }
  Vec2Cart& V()         { return _velocity; }

  std::string Color() const { return _color; }
  std::string& Color() { return _color; }
};
```

Next in line is container for our balls - simple rectangular box, with bottom left corner at point (0,0), and given width and height.

```
struct BoxContainer2D
{
  double _width;
  double _height;

  std::vector<Ball2D> _balls;

  BoxContainer2D() : _width(1000), _height(1000) {}

  void    AddBall(const Ball2D& body) { _balls.push_back(body); }
  Ball2D& Ball(int i) { return _balls[i]; }

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

Where CheckAndHandleOutOfBounds() function of the BoxContainer2D class 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;
  }
}
```

We'll need the utility class, SimResultsCollSim2D, containing results of performed simulations

```
class SimResultsCollSim2D
{
  int _numBalls;
public:
  std::vector<Ball2D>  Balls;                        // properties of balls
  std::vector<std::vector<Pnt2Cart>> BallPosList;    // positions of all balls at each time step
  std::vector<std::vector<Vec2Cart>> BallVelList;    // positions of all balls at each time step

  SimResultsCollSim2D(int numBalls){ ... }
  SimResultsCollSim2D(const std::vector<Ball2D>& balls){ ... }
  SimResultsCollSim2D(const std::vector<Ball2D>& balls, const std::vector<std::vector<Pnt2Cart>>& ballPositions){ ... }

  SimResultsCollSim2D(const std::vector<Ball2D>& balls, const std::vector<std::vector<Pnt2Cart>>& ballPositions,
    const std::vector<std::vector<Vec2Cart>>& ballVelocities){ ... }

  int NumBalls() const { return _numBalls; }

  const std::vector<Pnt2Cart>& getBallPositions(int indBall) const{ ... }
  const std::vector<Vec2Cart>& getBallVelocities(int indBall) const{ ... }

  std::vector<Pnt2Cart> getPathForBall(int ind) const{ ... }
```

Putting it all together we have CollisionSimulator2D class, implementing basic simulator.

```
class CollisionSimulator2D
{
  BoxContainer2D _box;
public:
  CollisionSimulator2D(const BoxContainer2D& box) : _box(box), M(2, 2) {}

  double DistBalls(int i, int j){ ... }
  bool   HasBallsCollided(int i, int j){ ... }
  void   HandleCollision(int m, int n, double dt){ ... }

  void SimulateOneStep(double dt){ ... }

  SimResultsCollSim2D Simulate(int numSteps, double timeStep){ ... }

  void Serialize(std::string fileName, const std::vector<std::vector<Pnt2Cart>>& ballPositions){ ... }
};
```

Where Simulate() function is quite trivial.

```
SimResultsCollSim2D Simulate(int numSteps, double timeStep)
{
   int       numBalls = _box._balls.size();
   double    dt = timeStep;

   SimResultsCollSim2D results(_box._balls);

   for (int i = 0; i < numSteps; i++)
   {
      // save positions for all balls at this step
      for (int j = 0; j < numBalls; j++)
      {
         results.BallPosList[j].push_back(_box._balls[j].Pos());
         results.BallVelList[j].push_back(_box._balls[j].V()); // save velocities too
      }
      SimulateOneStep(dt);
   }
   return results;
}
```

As is SimulateOneStep() function

```
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))
        HandleCollision(m, n, dt);
    }
  }
}
```

HandleCollision is the most important function here.

```
void HandleCollision(int m, int n, double dt)
{
  Ball2D& ball1 = _box._balls[m];
  Ball2D& ball2 = _box._balls[n];

  // calculating point where they were before collision
  // (if there was collision with the box wall, then calc.pos. will be outside the box
  // but it doesn't matter, since we need only direction, ie. velocity, to calculate exact collision point)
  Pnt2Cart x10 = ball1.Pos() - ball1.V() * dt;
  Pnt2Cart x20 = ball2.Pos() - ball2.V() * dt;

  Vec2Cart dx0(x10, x20);
  Vec2Cart dv(ball2.V() - ball1.V());

  // first, we have to calculate exact moment of collision, and balls positions then
  double A = dv * dv;
  double B = 2 * dx0 * dv;
  double C = dx0 * dx0 - POW2(ball2.Rad() + ball1.Rad());

  double t1 = (-B + sqrt(B * B - 4 * A * C)) / (2 * A);
  double t2 = (-B - sqrt(B * B - 4 * A * C)) / (2 * A);

  double tCollision = t1 < t2 ? t1 : t2;
```

```
    // calculating position of balls at the point of collision (moving them backwards)
    Pnt2Cart x1 = ball1.Pos() + (tCollision - dt) * ball1.V();
    Pnt2Cart x2 = ball2.Pos() + (tCollision - dt) * ball2.V();

    // 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);
}
```

PUTTING OUR COLLISIONSIMULATOR2D TO USE

First task in performing simulation is generating initial configuration, ie. generating N balls, and setting their initial positions and velocities.

For that, we have ContainerFactory2D class.

Set of functions for creating configurations for our investigations.

```
class ContainerFactory2D
{
public:
    // container with five different balls, with given masses, radii, colors, positions and velocities
    static BoxContainer2D CreateConfig(){ ... }

    // container with two types of balls, with masses, radii, positions and velocities
    // randomized in given intervals
    static BoxContainer2D CreateConfig1(double width, double height, int N,
        const std::string& color1, double minMass1, double maxMass1, double minRad1, double maxRad1, double velocityRange1,
        const std::string& color2, double minMass2, double maxMass2, double minRad2, double maxRad2, double velocityRange2 )

    // container with two types of balls, initially each occupies one half of the container
    // random balls position within each half, with given mass and velocity for each type
    static BoxContainer2D CreateConfig2(double width, double height,
        int nBall1, double mass1, double rad1, double vel1,
        int nBall2, double mass2, double rad2, double vel2){ ... }

    // container with a central big ball and many small balls around it, with random positions and velocities
    static BoxContainer2D CreateBrownMotionConfig(double width, double height, int numSmallBalls){ ... }

    // create a container with two types of balls, 20 of them are at the center and very energetic
    // the other 200 are scattered around the container with low energy
    static BoxContainer2D CreateConfig3(double width, double height,
                                        int numBalls, double mass1, double rad1, double vel1,
                                        int numEnergetic, double mass2, double rad2, double vel2 ){ ... }

    // container with equal Balls, N in line and evenly ditributed in the container with given width and height
    // total number of Balls is N*N, with given mass, radius and speed (given value, but random direction)
    static BoxContainer2D CreateConfig4(double width, double height, int N, double mass, double radius, double speed){ ..
};
```

Example 1 – mixing of two types of balls

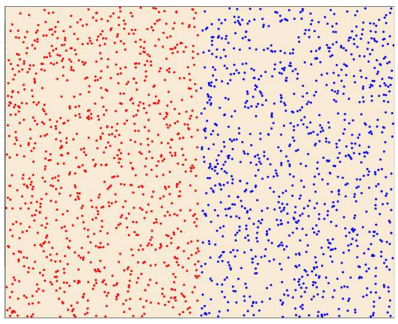
We have a container with a wall in the middle, separating balls with different properties. How will they mix once the wall is removed?

Creating initial configurations.

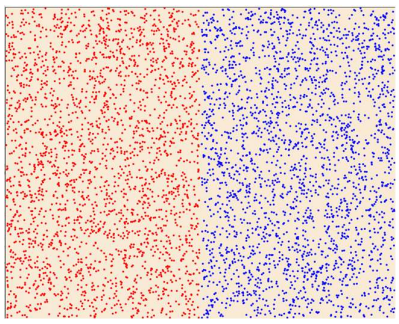
```
void Collision_Simulator_2D_two_types_initially_separated()
{
  int    nBall1 = 5000;
  double mass1 = 5, rad1 = 2, vel1 = 20;
  int    nBall2 = 5000;
  double mass2 = 5, rad2 = 2, vel2 = 20;

  auto box = ContainerFactory2D::CreateConfig2(1000, 800, nBall1, mass1, rad1, vel1, nBall2, mass2, rad2, vel2);
```

Example configurations:

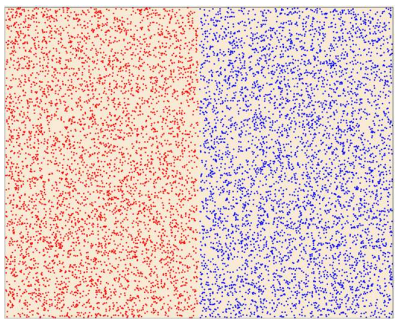
2.000 balls, radius = 3



5.000 balls, radius 2.5



10.000 balls, radius = 2



Performing simulation.

```
  CollisionSimulator2D simulator(box);

  int numSteps = 5;
  auto simResults = simulator.Simulate(numSteps, 0.1);

  simulator.Serialize(MML_PATH_ResultFiles + "collision_sim_2d_example1.txt", simResults.BallPosList)

  Visualizer::VisualizeParticleSimulation2D("collision_sim_2d_example1.txt");
}
```

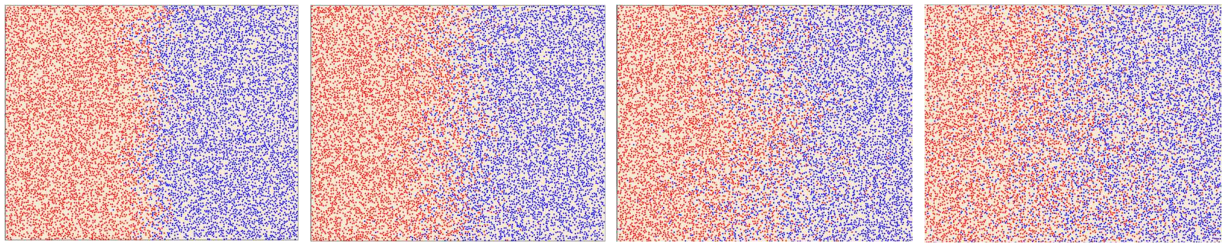
10.000 balls, with same and different initial velocities for blue balls, wit radius set to 2.5.

Situation after 50, 200, 500 and 1.000 “seconds”

v1 = 10 “m/s”.

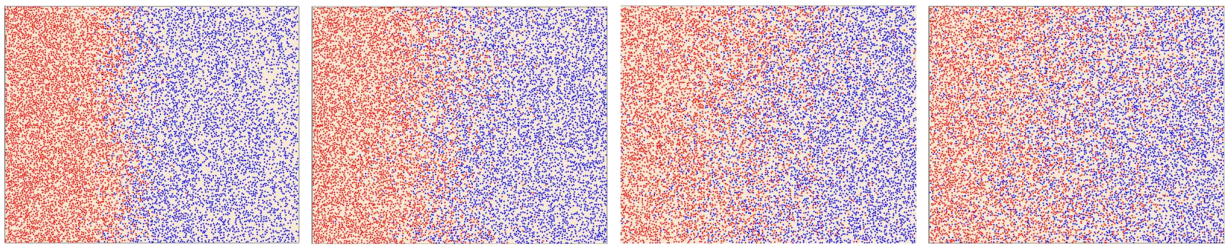
v2 = v1, dT = 0.1

50 sec 200 sec 500 sec 1.000 sec



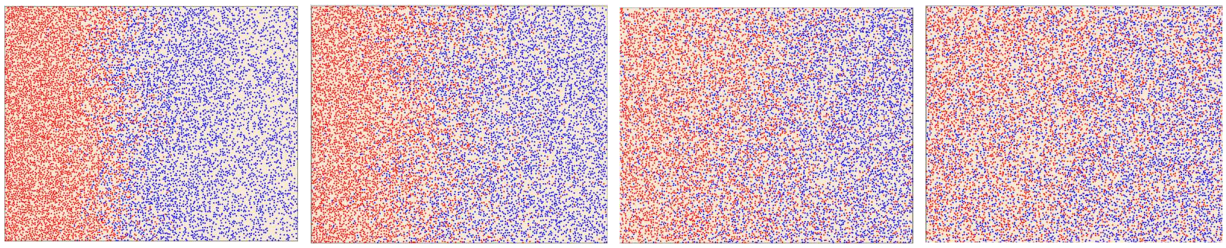
v2 = 2 * v1, dT = 0.1

50 sec 200 sec 500 sec 1.000 sec



v2 = 5 * v1, dT = 0.05

50 sec 200 sec 500 sec 1.000 sec



Example 2 – “explosion” of energetic balls in the middle of container

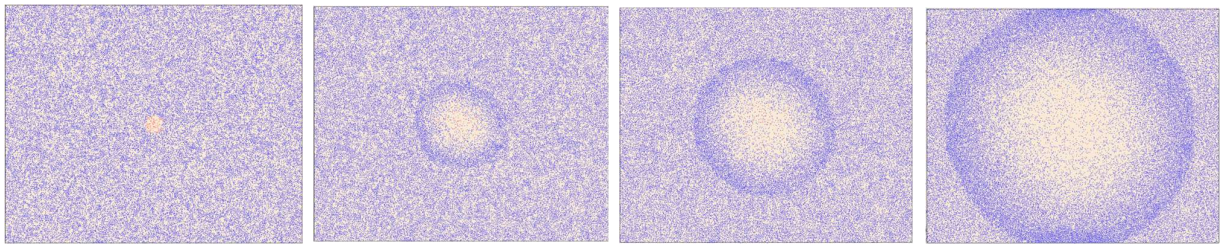
Setting up configuration with 50.000 balls, and 100 energetic balls in the middle.

```
int    numBalls = 50000;
double mass1 = 5, rad1 = 1, vel1 = 2;

int    numEnergetic = 100;
double mass2 = 5, rad2 = 1, vel2 = 500;

double posRadius = 25;

auto box = ContainerFactory2D::CreateConfig3(1000, 800, numBalls, mass1, rad1, vel1, numEnergetic, posRadius, mass2, rad2, vel2);
```

Performing simulation, and visualizing results

```
CollisionSimulator2D simulator(box, 10, 10);

int numSteps = 1000;
double dT = 0.005;

auto simResults = simulator.Simulate(numSteps, dT);

simulator.Serialize(MML_PATH_ResultFiles + "collision_sim_2d_3.txt", simResults, dT);

Visualizer::VisualizeParticleSimulation2D("collision_sim_2d_3.txt");
```

Step = 0 Step = 250 Step = 500 Step = 1000



ADDING BASIC STATISTICS FOR FOLLOWING BALLS EVOLUTION

In the third case, with v2 = 5 * v1, it is observed that blue balls “compress” red balls in the left of the container, where at one moment they occupy only quarter of the container, and after that we have a “wave” of red balls in opposite directions.

How could we quantify and measure that behavior?

The answer is – by extending SimResults class that has all the data that we calculate during simulation

We have two sets of statistics that we can calculate for every step of our solution.

First set is calculated for all balls

```
// Statistics for all balls
void CalcAllBallsStatistic(int timeStep, double& minSpeed, double& maxSpeed, double& avgSpeed, double& speedDev)

Pnt2Cart CentreOfMass(int timeStep) { ... }

Pnt2Cart AvgPos(int timeStep) { ... }

Real AvgSpeed(int timeStep) { ... }

Real TotalKineticEnergy(int timeStep) { ... }
```

And the second one divides balls by type (ie. “color”) and calculates statistics separately for every color, giving insight into differences in behavior for different types of balls.

```
// Statistics by color
// for given time step, returns vector of pairs (color, center of mass)
std::vector<std::pair<std::string, Pnt2Cart>> CentreOfMassByColor(int timeStep) { ... }

std::vector<std::pair<std::string, Pnt2Cart>> AvgPosByColor(int timeStep) { ... }

std::vector<std::pair<std::string, Real>> AvgSpeedByColor(int timeStep) { ... }

std::vector<std::pair<std::string, Real>> TotalKineticEnergyByColor(double timeStep) { ... }
```

Using it to observe change in average speed of balls in third case, with v2 = 5 * v1

Setup of simulation

```
int    nBall1 = 5000;
double mass1 = 5, rad1 = 2.0, vel1 = 10;
int    nBall2 = 5000;
double mass2 = 5, rad2 = 2.0, vel2 = 50;    // five times faster than red balls

auto box = ContainerFactory2D::CreateConfig2(1000, 800, "Red", nBall1, mass1, rad1, vel1, "Blue", nBall2, mass2, rad2, vel2);

CollisionSimulator2D simulator(box);

int numSteps = 5000;
double dT = 0.1;                    // time step for simulation
double totalTime = numSteps * dT;   // maximum time for simulation

auto simResults = simulator.Simulate(numSteps, dT);
```

Extracting data from SimResults class.

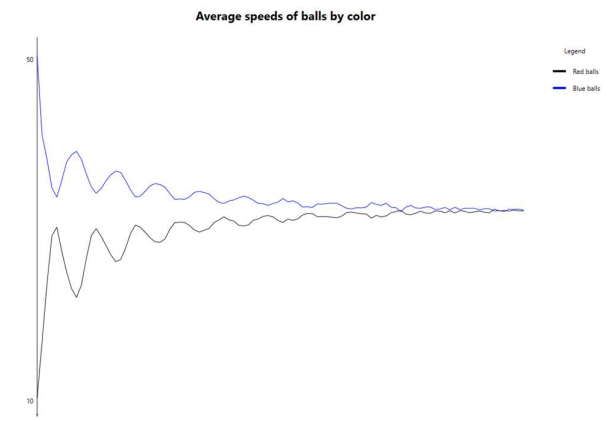
```
Vector<Real> vecTime, avgSpeedRed, avgSpeedBlue;
for (int i = 0; i < numSteps; i += 10)
{
  auto avgs = simResults.AvgSpeedByColor(i);

  if( avgs[0].first == "Red" ) {
    avgSpeedRed.push_back(avgs[0].second);
    avgSpeedBlue.push_back(avgs[1].second);
  }
  else {
    avgSpeedRed.push_back(avgs[1].second);
    avgSpeedBlue.push_back(avgs[0].second);
  }
  vecTime.push_back(i * dT);
}
```

Visualizing

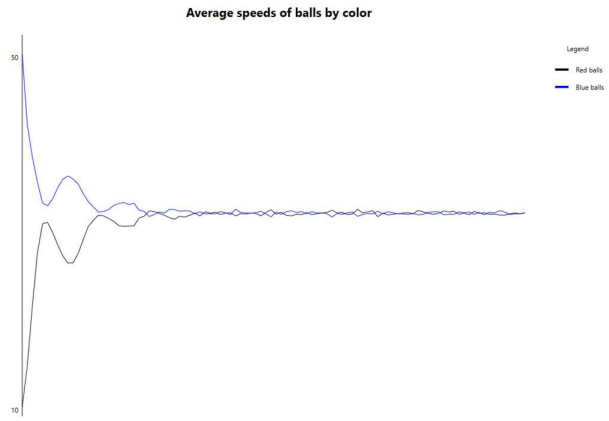
```
LinearInterpRealFunc avgSpeedRedFunc(vecTime, avgSpeedRed);
LinearInterpRealFunc avgSpeedBlueFunc(vecTime, avgSpeedBlue);

Visualizer::VisualizeMultiRealFunction({ avgSpeedRedFunc, avgSpeedBlueFunc },
                                       "Average speeds of balls by color", { "Red balls", "Blue balls" },
                                       vecTime[0], vecTime[vecTime.size() - 1], 100,
                                       "avg_speed_by_color.txt");
```

Results, with performed 5.000 steps

Balls radius = 2

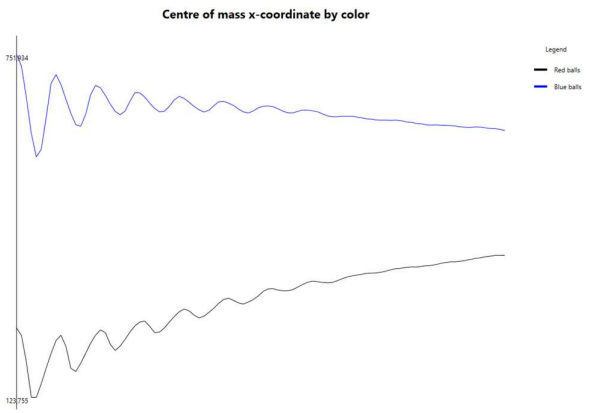


Balls radius = 1

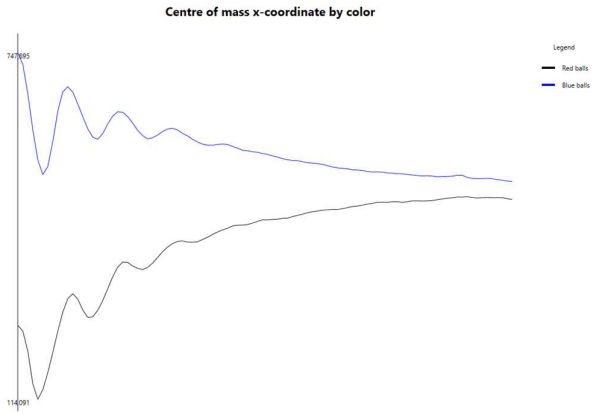


We can also visualize movement of the center of mass of balls of both types. And, as the container is divided along y-axis, relevant is change in x-component of CoM.

Balls radius = 2



Balls radius = 1



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?

We will implement two approaches:

1) Partition the space of our container so we don't have to check for collisions between all balls at each step
2) Employ multithreading to evaluate collision in subdivided compartments of container in parallel

Implementing space partitioning

```
  // M is a matrix of vectors, where each vector contains indices of balls in that subcontainer
  Matrix<Vector<int>> M;

public:
  CollisionSimulator2D() : M(2, 2) {}
  CollisionSimulator2D(const BoxContainer2D& box) : _box(box), M(2, 2) {}
  CollisionSimulator2D(const BoxContainer2D& box, int nRows, int nCols) : _box(box), M(nRows, nCols) {}
```

Changes in BoxContainer2D.

We need to add following function.

```
// fill given matrix with the number of balls in each subdivided container
void SetNumBallsInSubdividedContainer(int nRows, int nCols, Matrix<Vector<int>>& M)
{
  double cellWidth = _width / nCols;
  double cellHeight = _height / nRows;

  for (int i = 0; i < _balls.size(); i++)
  {
    Ball2D& ball = _balls[i];

    int row = static_cast<int>(ball.Pos().Y() / cellHeight);
    int col = static_cast<int>(ball.Pos().X() / cellWidth);

    if (row >= 0 && row < nRows && col >= 0 && col < nCols)
    {
      M(row, col).push_back(i);
    }
  }
}
```

That we'll add SimulateOneStepFast() function that uses space partition

```
void SimulateOneStepFast(double dt)
{
  int NumBalls = _box._balls.size();

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

  _box.SetNumBallsInSubdividedContainer(M.RowNum(), M.ColNum(), M);

  // handle out of bounds situations
  for (int i = 0; i < NumBalls; i++)
    _box.CheckAndHandleOutOfBounds(i);

  // now, we have to check for collisions only in subcontainers where there are balls
  for (int i = 0; i < M.RowNum(); i++)  {
    for (int j = 0; j < M.ColNum(); j++)
    {
      if (M(i, j).size() > 1) // if there are at least two balls in this subcontainer
      {
        for (int m = 0; m < M(i, j).size() - 1; m++) {
          for (int n = m + 1; n < M(i, j).size(); n++)
          {
            int ballIndex1 = M(i, j)[m];
            int ballIndex2 = M(i, j)[n];

            if (HasBallsCollided(ballIndex1, ballIndex2))
              HandleCollision(ballIndex1, ballIndex2, dt);
          }
        }
        M(i, j).Clear();  // we are done with this subcontainer, clear it for next iteration
      }
    }
  }
}
```

Implementing multi-threading

Thanks to impressive advances in multithreading support in C++ Standard, it is quite trivial to extend our SimulateOneStepFasFast() function to employ multithreading

Minimal change to our loop on subdivisions of the container:

```
std::vector<std::thread> threads;
for (int i = 0; i < M.RowNum(); i++)
{
  for (int j = 0; j < M.ColNum(); j++)
  {
    if (M(i, j).size() > 1) // if there are at least two balls in this subcontainer
    {
      threads.emplace_back([this, i, j, dt]() {
          for (int m = 0; m < M(i, j).size() - 1; m++) {
            for (int n = m + 1; n < M(i, j).size(); n++)
            {
              int ballIndex1 = M(i, j)[m];
              int ballIndex2 = M(i, j)[n];

              if (HasBallsCollided(ballIndex1, ballIndex2))
                HandleCollision(ballIndex1, ballIndex2, dt);
            }
          }
          M(i, j).Clear();
        });
    }
  }
}
for (auto& t : threads) t.join();
```

Implementing thread-pooling

With larger number of subdivisions, number of created compartments, each running in its own thread grows as N^2 and creating immediately all those threads and starting them is not an optimal execution strategy

We create class ThreadPool:

```
class ThreadPool {
private:
  std::vector<std::thread> workers;
  std::queue<std::function<void()>> tasks;
  std::mutex queue_mutex;
  std::condition_variable condition;
  bool stop;
```

Where the most important part of the class is the constructor:

```
public:
  ThreadPool(size_t numThreads) : stop(false)
  {
    for (size_t i = 0; i < numThreads; ++i)
      workers.emplace_back([this]
        {
          while (true) {
            std::function<void()> task;
            {
              std::unique_lock<std::mutex> lock(this->queue_mutex);

              this->condition.wait(lock, [this] { return this->stop || !this->tasks.empty(); });

              if (this->stop && this->tasks.empty())
                return;

              task = std::move(this->tasks.front());
              this->tasks.pop();
            }
            task();
          }
        });
  }
```

And implementation of SimulateOneStepFastMultithreadThreadPool() function:

Beginning is the same:

```
_box.SetNumBallsInSubdividedContainer(M.RowNum(), M.ColNum(), M);

int NumBalls = _box._balls.size();

// Update 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);
}
```

But then we need to setup our thread pool.

```
unsigned int numThreads = std::thread::hardware_concurrency();
if (numThreads == 0)
  numThreads = 2; // Fallback if detection fails

ThreadPool pool(numThreads);

std::atomic<int> tasksRemaining = 0;
```

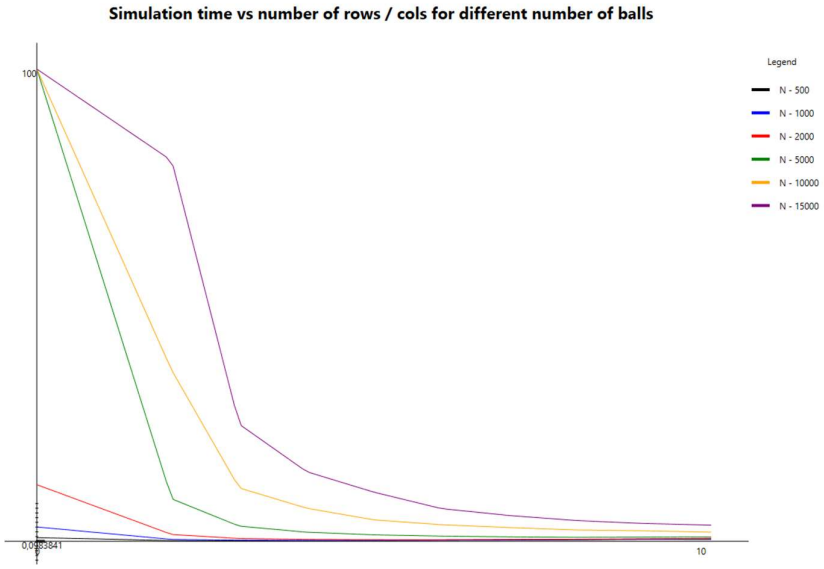
The main loop where we add all calculations as separate tasks to ThreadPool

```
for (int i = 0; i < M.RowNum(); i++) {
  for (int j = 0; j < M.ColNum(); j++) {
    if (M(i, j).size() > 1) {
      tasksRemaining++;
      pool.enqueue([this, i, j, dt, &tasksRemaining]()
        {
          for (int m = 0; m < M(i, j).size() - 1; m++) {
            for (int n = m + 1; n < M(i, j).size(); n++)
            {
              int ballIndex1 = M(i, j)[m];
              int ballIndex2 = M(i, j)[n];
              if (HasBallsCollided(ballIndex1, ballIndex2))
                HandleCollision(ballIndex1, ballIndex2, dt);
            }
          }
          M(i, j).Clear();
          --tasksRemaining;
        });
    }
  }
}
```

And final cleanup after loop.

```
// Wait for all tasks to finish
while (tasksRemaining > 0) {
    std::this_thread::yield();
}
```

Testing efficiency improvements



ADDING MEASUREMENT OF PRESSURE ON THE WALLS

```
// represents single collision with wall
struct WallCollisionEvent2D
{
  int   _ballInd;
  Real  _transferedMomentum; // momentum transferred to the wall by the ball

  WallCollisionEvent2D(int ballInd = -1, Real transferedMomentum = 0.0) { ... }
};
```

```
class Wall
{
  std::string _name;    // name of the wall, e.g. "left", "right", "bottom", "top"
public:
  Wall() : _name("") {}
  Wall(const std::string& name) : _name(name) {}

  const std::string& Name() const { return _name; }
};
```

Interface

```
class IWallPressureRecorder2D
{
public:
  virtual void SetNumberOfWalls(int nWalls) = 0;
  virtual void StartNextStep() = 0;
  virtual void RecordPressureEvent(int wallIndex, const WallCollisionEvent2D& event) = 0;
};
```

And class to tie it all together (for our BoxContainer2D!)

```
class BoxWallPressureRecorder2D : public IWallPressureRecorder2D
{
  std::vector<Wall> _walls;

  // collision events for each wall, indexed first by step index, then by wall index
  std::vector<std::vector<std::vector<WallCollisionEvent2D>>> _collisionEvents;

  int _currStep;

public:
  BoxWallPressureRecorder2D(int expectedSteps = 100)
    : _currStep(0)
  {
    _collisionEvents.resize(expectedSteps);

    _walls.resize(4); // left, right, bottom, top
    for (int i = 0; i < expectedSteps; i++)
      _collisionEvents[i].resize(4);
  }

  void SetNumberOfWalls(int nWalls) { ... }
  void StartNextStep() { ... }
  void RecordPressureEvent(int wallIndex, const WallCollisionEvent2D& event) { ... }

  // get all events for wall
  std::vector<std::vector<WallCollisionEvent2D>> getEventsPerStepForWall(int wallIndex) { ... }

  Vector<Real> getTranferedMomentumPerStepForWall(int wallIndex) { ... }
};
```

Changes in CollisionSimulator are minimal, added member, with constructors to initialize it

```
// for recording wall pressure events, if needed
IWallPressureRecorder2D* _wallPressureRecorder = nullptr;

public:
// ...
CollisionSimulator2D(const BoxContainer2D& box, IWallPressureRecorder2D* wallPressureRecorder) { ... }
CollisionSimulator2D(const BoxContainer2D& box, int nRows, int nCols, IWallPressureRecorder2D* wallPressureRecorder)

void setWallPressureRecorder(IWallPressureRecorder2D* wallPressureRecorder) { ... }
```

And a change in Simulate function, to mark start of the next step

```
SimResultsCollSim2D Simulate(int numSteps, double timeStep, bool useFast = true)
{
  int     numBalls = _box._balls.size();
  double  dt = timeStep;

  SimResultsCollSim2D results(_box._balls);

  for (int i = 0; i < numSteps; i++)
  {
    if (_wallPressureRecorder != nullptr)
      _wallPressureRecorder->StartNextStep();

    // save positions for all balls at this step
    for (int j = 0; j < numBalls; j++)
```

As the BoxContainer2D class is responsible for handling collision, and has all the data, there are changes there also.

It also needs a reference to WallPressureRecorder

```
IWallPressureRecorder2D* _wallPressureRecorder = nullptr; // for recording wall pressure events, if needed

BoxContainer2D() : _width(1000), _height(1000) {}
BoxContainer2D(double width, double height) : _width(width), _height(height) {}
BoxContainer2D(double width, double height, IWallPressureRecorder2D* wallPressureRecorder) { ... }

void setWallPressureRecorder(IWallPressureRecorder2D* wallPressureRecorder) { ... }
```

And in the presence of WallPressureRecorder, handling out of bounds is different, meaning, we are calculating transferred momentum to wall and add a PressureEvent

```
void CheckAndHandleOutOfBounds(int ballIndex)
{
  Ball2D& ball = _balls[ballIndex];

  // 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;
    if (_wallPressureRecorder)
    {
      // record pressure event for the left wall
      Real transferedMomentum = ball.Mass() * fabs(ball.V().X());
      _wallPressureRecorder->RecordPressureEvent(0, WallCollisionEvent2D(ballIndex, transferedMomentum));
    }
  }
  // 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;

    if (_wallPressureRecorder)
    {
      // record pressure event for the right wall
      Real transferedMomentum = ball.Mass() * fabs(ball.V().X());
      _wallPressureRecorder->RecordPressureEvent(1, WallCollisionEvent2D(ballIndex, transferedMomentum));
    }
  }
  // bottom wall collision
  if (ball.Pos().Y() < ball.Rad() && ball.V().Y() < 0)
  {
```

Analyzing and visualizing pressure on walls during simulation

With all this in place, we can easily gather data about pressure on the walls during simulation.

Setting up configuration

```
int numBalls = 10000;

const std::string& color1 = "Red";
double minMass1 = 1.0, maxMass1 = 10.0, minRad1 = 1.0, maxRad1 = 2.0, velocityRange1 = 5.0;
const std::string& color2 = "Blue";
double minMass2 = 1.0, maxMass2 = 10.0, minRad2 = 1.0, maxRad2 = 2.0, velocityRange2 = 5.0;

auto box = ContainerFactory2D::CreateConfig1(1000, 800, numBalls,
                                        color1, minMass1, maxMass1, minRad1, maxRad1, velocityRange1,
                                        color2, minMass2, maxMass2, minRad2, maxRad2, velocityRange2);
```

Performing simulation

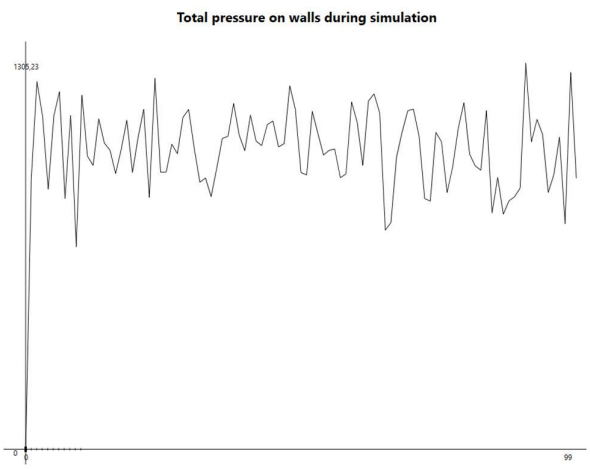
```
BoxWallPressureRecorder2D recorder(numBalls);
box.setWallPressureRecorder(&recorder);

CollisionSimulator2D simulator(box, 8, 8, &recorder);

int numSteps = 500;
double dT = 1;
auto simResults = simulator.Simulate(numSteps, dT, true);
```

Analyzing and visualizing results:

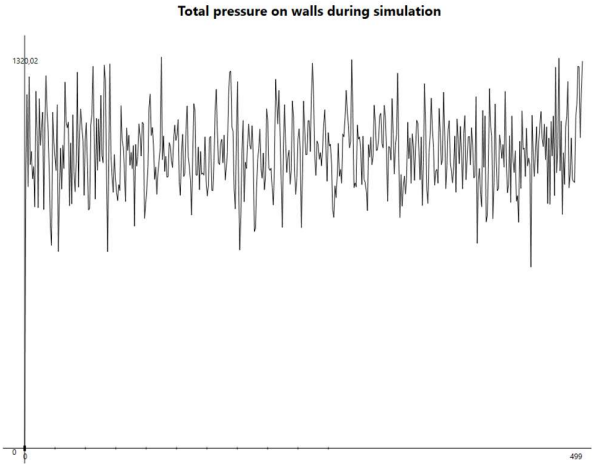
```
auto totalWallPressure = recorder.getTotalTransferedMomentumPerStep();

Vector<Real> x, y;
for (int i = 0; i < totalWallPressure.size(); i++)
{
  x.push_back(i);
  y.push_back(totalWallPressure[i]);
}
LinearInterpRealFunc funcTotalPressure(x, y);

// visualize function of total pressure on walls
Visualizer::VisualizeRealFunction(funcTotalPressure, "Total pressure on walls during simulation",
                                  0, numSteps - 1, numSteps-1, "total_pressure_on_walls.txt");
```

Resulting graph of total pressure.

Num steps = 100



Num steps = 500



Dependence of pressure on N (number of balls)

Setup

```
int numSteps = 500;

Vector<int>  numBallsList{ 1000, 2000, 4000, 8000, 16000 };
Vector<Real> avgPressure;

std::vector<std::string> strNumBalls;
std::vector<LinearInterpRealFunc> totalPressureFuncs;
```

Loop for iterating over different values for number of balls has three parts:

Setup

```
for (int i=0; i<numBallsList.size(); i++)
{
  int numBalls = numBallsList[i];

  const std::string color = "Red";
  double mass = 1.0, rad = 2.0, velocityRange = 5.0;

  auto box = ContainerFactory2D::CreateConfigNSameBalls(1000, 1000, numBalls,
                                                        mass, rad, velocityRange, color);
```

Simulation

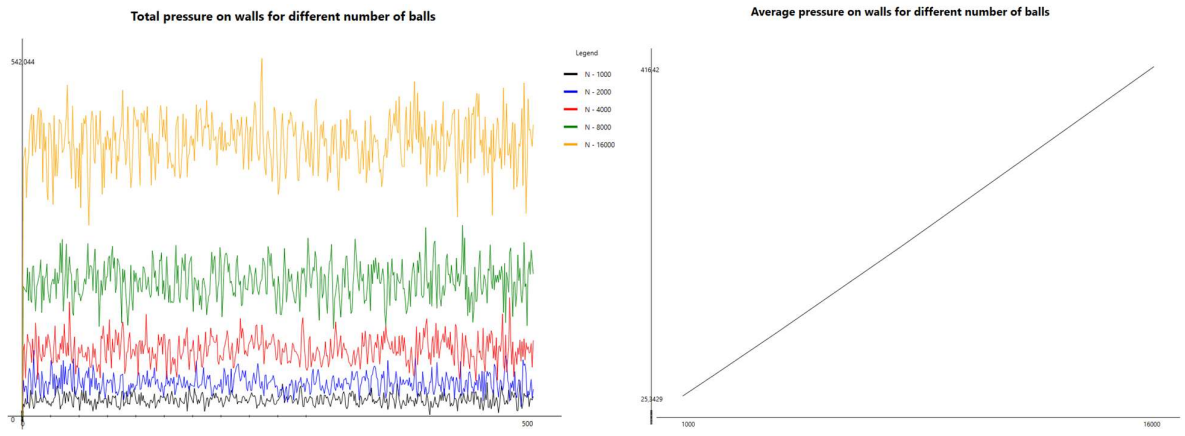
Little bit more involved as we want more space subdivisions for higher number of balls in simulation.

```
BoxWallPressureRecorder2D recorder(numBalls);
box.setWallPressureRecorder(&recorder);

CollisionSimulator2D *pSimulator;
if( numBalls < 10000 )
  pSimulator = new CollisionSimulator2D(box, 4, 4, &recorder);
else
  pSimulator = new CollisionSimulator2D(box, 8, 8, &recorder);

auto simResults = pSimulator->Simulate(numSteps, 1.0, true);
```

Analysis and visualization

```
  auto totalWallPressure = recorder.getTotalTransferedMomentumPerStep();

  Vector<Real> x, y;
  for (int i = 0; i < totalWallPressure.size(); i++)
  {
    x.push_back(i);
    y.push_back(totalWallPressure[i]);
  }

  LinearInterpRealFunc funcTotalPressure(x, y);

  totalPressureFuncs.push_back(funcTotalPressure);
  strNumBalls.push_back("N - " + std::to_string(numBallsList[i]));

  delete pSimulator;    // delete simulator to free memory
}

// visualize all total pressure functions on one plot
Visualizer::VisualizeMultiRealFunction(totalPressureFuncs,
                                       "Total pressure on walls for different number of balls", strNumBalls,
                                       0, numSteps, numSteps, "total_pressure_multi_N.txt");
```

Results:



Where linear dependence is obvious.

Dependence of pressure on V (volume of the container)

Initial setup is a little bit different

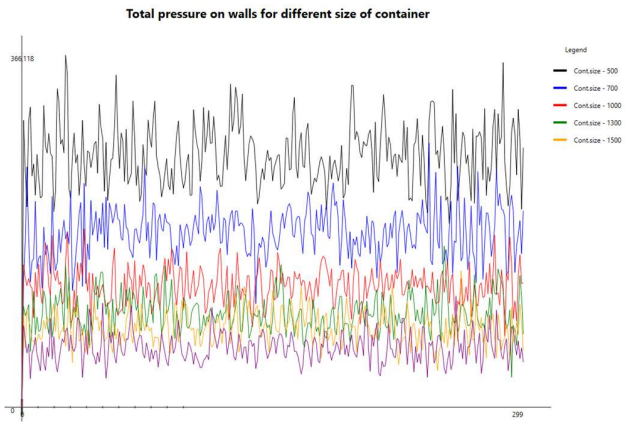
```
int numSteps = 300;
int numBalls = 2000;    // number of balls in each simulation

Vector<int>  containerSizeList{ 500, 700, 1000, 1300, 1500, 2000 };
Vector<Real> avgPressure;

std::vector<std::string> strContSize;
std::vector<LinearInterpRealFunc> totalPressureFuncs;
```

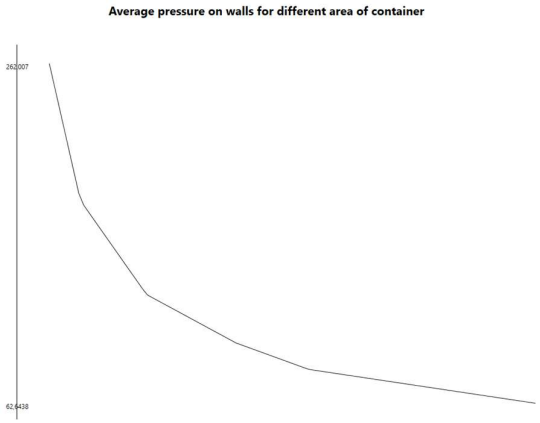
And we actually have to take container size squared to get the “volume” (in 2D case it is area)

```
Vector<Real> x_squared;
for (int i = 0; i < containerSizeList.size(); i++)
  x_squared.push_back(containerSizeList[i]*containerSizeList[i]);
```

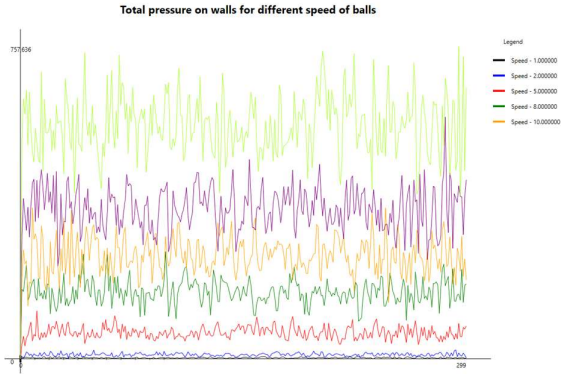
```
LinearInterpRealFunc avgPressureFunc(x_squared, avgPressure);

double maxX = POW2(containerSizeList[containerSizeList.size() - 1]);
Visualizer::VisualizeRealFunction(avgPressureFunc, "Average pressure on walls for different area of container",
                                  x_squared[0], maxX, 100, "avg_pressure_on_walls_diff_V.txt");
```

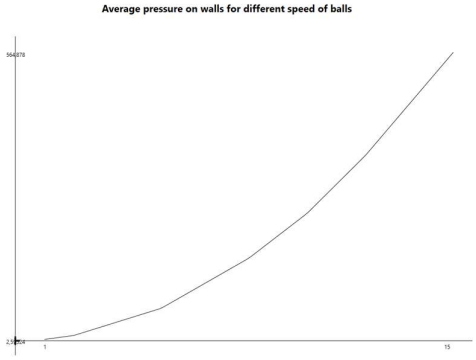
Results:

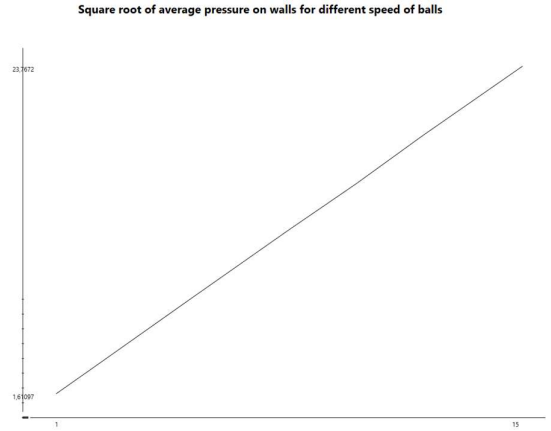




Dependence of pressure on T (speed of the balls)







IMPLEMENTING 3D SIMULATION

After implementing complete simulator for 2D case, adjustment for 3D simulation is relatively easy, with just some necessary tweaks.

First, obviously, Pnt3Cart and Vec3Cart are used instead of Pnt2Cart and Vec2Cart.

Then we have Ball3D, which is trivial, and BoxContainer3D, which requires two adjustments.

First, we have to adjust CheckAndHandleOutOfBonds() for 3D case, and second, instead of a simple Matrix, use Matrix3D class for handling data about balls in subdivided compartments of the container.

```
void CheckAndHandleOutOfBounds(int ballIndex)
{
  Ball3D& ball = _balls[ballIndex];

  if (ball.Pos().X() < ball.Rad() && ball.V().X() < 0)    // checking if ball is out of bounds
  {
    ball.Pos().X() = ball.Rad() + (ball.Rad() - ball.Pos().X());    // moving it back to box
    ball.V().X() *= -1;
  }

  if (ball.Pos().X() > _width - ball.Rad() && ball.V().X() > 0)
  {
    ball.Pos().X() -= (ball.Pos().X() + ball.Rad()) - _width;
    ball.V().X() *= -1;
  }

  if (ball.Pos().Y() < ball.Rad() && ball.V().Y() < 0)
  {
    ball.Pos().Y() = ball.Rad() + (ball.Rad() - ball.Pos().Y());
    ball.V().Y() *= -1;
  }

  if (ball.Pos().Y() > _height - ball.Rad() && ball.V().Y() > 0)
  {
    ball.Pos().Y() -= (ball.Pos().Y() + ball.Rad()) - _height;
    ball.V().Y() *= -1;
  }

  if (ball.Pos().Z() < ball.Rad() && ball.V().Z() < 0)
  {
    ball.Pos().Z() = ball.Rad() + (ball.Rad() - ball.Pos().Z());
    ball.V().Z() *= -1;
  }

  if (ball.Pos().Z() > _depth - ball.Rad() && ball.V().Z() > 0)
  {
    ball.Pos().Z() -= (ball.Pos().Z() + ball.Rad()) - _depth;
    ball.V().Z() *= -1;
  }
}
```

Adding space partitioning and multi-threading in 3D

Implementing multithreading support in 3D is exactly the same as in the 2D case, but for handling subdivisions of space, we need adjustments - we need to employ Matrix3D class in our BoxContainer3D class, with appropriate function for setting balls in subdivision compartments.

```
void SetNumBallsInSubdividedContainer(int n1, int n2, int n3, Matrix3D<Vector<int>>& M)
{
  double cellWidth = _width / n1;
  double cellHeight = _height / n2;
  double cellDepth = _depth / n3;

  for (int i = 0; i < _balls.size(); i++)
  {
    Ball3D& ball = _balls[i];

    int row = static_cast<int>(ball.Pos().Y() / cellHeight);
    int col = static_cast<int>(ball.Pos().X() / cellWidth);
    int depth = static_cast<int>(ball.Pos().Z() / cellDepth);

    if (row >= 0 && row < n1 && col >= 0 && col < n2 && depth >= 0 && depth < n3)
    {
      M(row, col, depth).push_back(i);
    }
  }
}
```

Adjusting classes for simulation results and pressure recording is trivial, and we arrive at our CollisionSimulator3D class, that is much similar to 2D case.

```
class CollisionSimulator3D
{
  BoxContainer3D _box;

  // 3D matrix for subdividing space into subcontainers, for faster collision detection
  Matrix3D<Vector<int>> M;

  // for recording wall pressure events, if needed
  IWallPressureRecorder3D* _wallPressureRecorder = nullptr;

public:
  CollisionSimulator3D() : M(2,2,2) {}
  CollisionSimulator3D(const BoxContainer3D& box) : _box(box), M(2, 2, 2) {}
  CollisionSimulator3D(const BoxContainer3D& box, int n1, int n2, int n3) : _box(box), M(n1, n2, n3) {}
  CollisionSimulator3D(const BoxContainer3D& box, IWallPressureRecorder3D* wallPressureRecorder) { ... }
  CollisionSimulator3D(const BoxContainer3D& box, int n1, int n2, int n3, IWallPressureRecorder3D* wallPressureRecorder)

  void setWallPressureRecorder(IWallPressureRecorder3D* wallPressureRecorder) { ... }

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

  void HandleCollision(int m, int n, double dt) { ... }

  void SimulateOneStepExact(double dt) { ... }
  void SimulateOneStepFast(double dt) { ... }
  void SimulateOneStepFastMultithread(double dt) { ... }

  SimResultsCollSim3D Simulate(int numSteps, double timeStep, bool useFast = true) { ... }

  bool Serialize(std::string fileName, const SimResultsCollSim3D& simResults, double dT, int saveEveryNSteps = 1) { ...
};
```

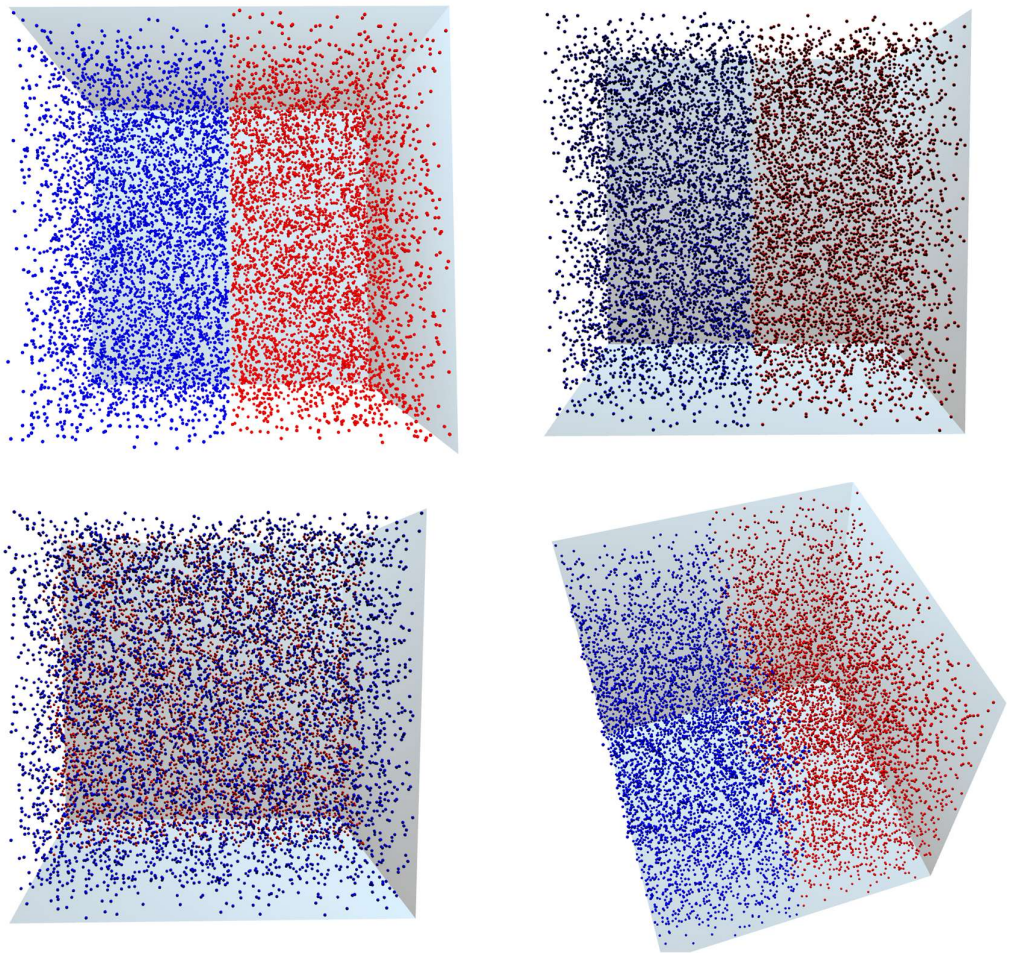
Example 1 – mixing of two types of balls in 3D

Setting up simulation is simple

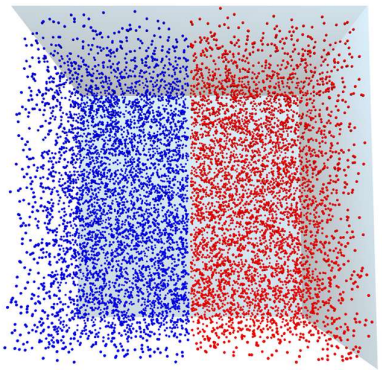
```
int    nBall1 = 5000;
double mass1 = 5, rad1 = 4, vel1 = 10;
int    nBall2 = 5000;
double mass2 = 5, rad2 = 4, vel2 = 10;

auto box = ContainerFactory3D::CreateConfig2(1000, 1000, 1000,
                                             nBall1, mass1, rad1, vel1,
                                             nBall2, mass2, rad2, vel2);
```

And running it and visualizing also.

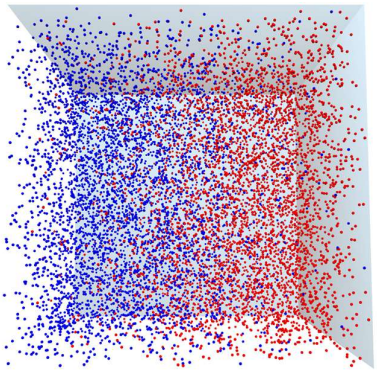
```
CollisionSimulator3D simulator(box, 3, 3, 3);

int  numSteps = 100;
double dT = 0.5;
auto simResult = simulator.Simulate(numSteps, dT);

Serializer::SaveParticleSimulation3D(MML_PATH_ResultFiles + "collision_sim_3d_example2.txt",
                                     box._balls.size(), box._width, box._height, box._depth,
                                     simResult.BallPosList, box.getBallColors(), box.getBallRadii(), dT);

Visualizer::VisualizeParticleSimulation3D("collision_sim_3d_example2.txt");
```



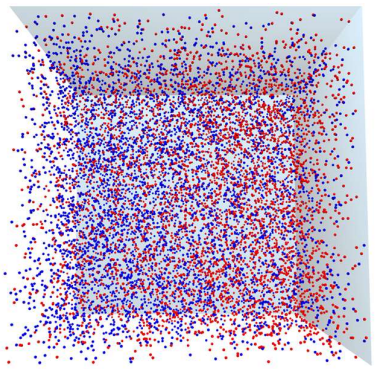
Start



After 100 steps



After 250 steps



Example 2 – simulating air

Simulating air properly, with necessary scaling.

Cube is 100 nm of length along each dimension, and within that volume, at STP conditions, there are 29.000 molecules of air in that volume.

```
// for 100 nm at STP, we get 27.000 molecules of ideal gas!!!
double boxSide_nm = 100.0;
double boxSide = boxSide_nm * 1e-9;
double px_per_nm = 10;                  // 10 pixels per nm, so 1 pixel = 0.1 nm

// STP constants
double k_B = PhyConst::BoltzmannConstant;   // Boltzmann constant in J/K
double T = 273.0;             // temperature in K
double P = 101325.0;          // pressure in Pa (1 atm)
```

```
const GasData& O2 = GasConstants::gasByFormula("O2");
const GasData& N2 = GasConstants::gasByFormula("N2");
const GasData& Ar = GasConstants::gasByFormula("Ar");
const GasData& CO2 = GasConstants::gasByFormula("CO2");

// oxygen (O2) gas at 273 K
double radius_O2_nm        = O2.RadiusKinetic_nm();
double molar_mass_O2       = O2.molarMass_g_mol / 1000;     // molar mass in kg/mol
double mass_molecule_O2_kg = O2.MoleculeMass_kg();
double avg_speed_O2_m_s    = IdealGasCalculator::AvgSpeed(molar_mass_O2, T);
```

```
// calculate TOTAL number of molecules in a 100 nm cube
double volume_m3 = boxSide * boxSide * boxSide;
double numBalls = (int) P * volume_m3 / (k_B * T); // number of molecules using ideal gas law
```

```
double rad_O2 = radius_O2_nm * px_per_nm;
double rad_N2 = radius_N2_nm * px_per_nm;
double rad_Ar = radius_Ar_nm * px_per_nm;
double rad_CO2 = radius_CO2_nm * px_per_nm;

double avg_speed_O2_px = avg_speed_O2_m_s * px_per_nm;    // in pixels/s
double avg_speed_N2_px = avg_speed_N2_m_s * px_per_nm;    // in pixels/s
double avg_speed_Ar_px = avg_speed_Ar_m_s * px_per_nm;    // in pixels/s
double avg_speed_CO2_px = avg_speed_CO2_m_s * px_per_nm;  // in pixels/s

int boxSide_px = boxSide_nm * px_per_nm;          // in pixels, so 100 nm in real world
```

We'll use config function

```
// container with four types of balls, representing gases in air - nitrogen, oxygen, argon and carbon dioxide
static BoxContainer3D CreateConfigGases(double width, double height, double depth, int totalMolecules,
                                        double rad_N2, double rad_O2, double rad_Ar2, double rad_CO2,
                                        double vel_N2, double vel_O2, double vel_Ar2, double vel_CO2,
                                        std::string color_N2, std::string color_O2, std::string color_Ar2, std::string color_CO2 )
{
  // ...
  int nitrogenCount = static_cast<int>(totalMolecules * 0.78);
  int oxygenCount = static_cast<int>(totalMolecules * 0.21);
  int argonCount = static_cast<int>(totalMolecules * 0.0093);
  int carbonDioxideCount = static_cast<int>(totalMolecules * 0.0004);
```

That adds randomly positioned balls/molecules for all four gasses in similar fashin:

```
BoxContainer3D box(width, height, depth);
// Nitrogen
for (int i = 0; i < nitrogenCount; i++)
{
  Pnt3Cart position(Random::UniformReal(rad_N2, width - rad_N2), Random::UniformReal(rad_N2, height - rad_N2),
                    Random::UniformReal(rad_N2, depth - rad_N2));

  double vx, vy, vz;
  Random::UniformVecDirection3(vx, vy, vz, vel_N2);
  Vec3Cart velocity(vx, vy, vz);

  box.AddBall(Ball3D(0.02802, rad_N2, color_N2, position, velocity));
}
```

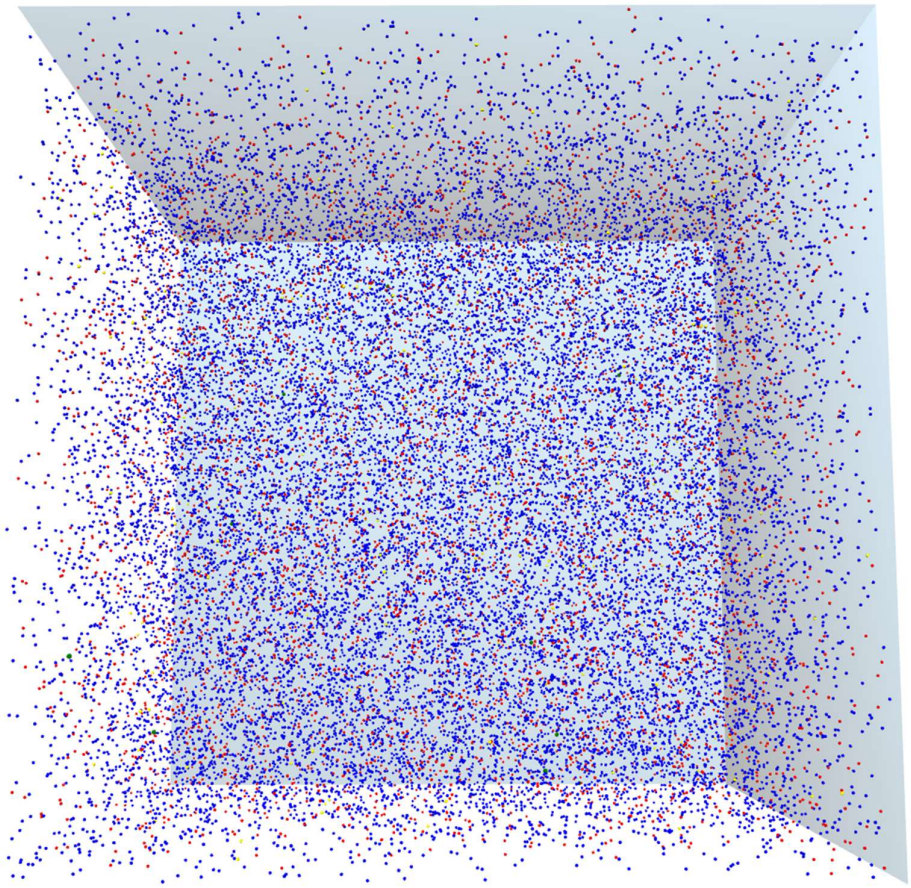
Our simulation, where we create configuration, simulate and visualize results

```
auto box = ContainerFactory3D::CreateConfigGases( boxSide_px, boxSide_px, boxSide_px, numBalls,
                                                  rad_N2, rad_O2, rad_Ar, rad_CO2,
                                                  avg_speed_N2_px, avg_speed_O2_px, avg_speed_Ar_px, avg_speed_CO2_px,
                                                  "Blue", "Red", "Yellow", "Green");
int numSteps = 10;        // number of timesteps
double dT = 0.001;        // in ns, so 0.001 ns per timestep

CollisionSimulator3D simulator(box, 5, 5, 5);

auto simResult = simulator.Simulate(numSteps, dT);

Serializer::SaveParticleSimulation3D(MML_PATH_ResultFiles + "air_as_ideal_gas.txt", box._balls.size(),
                                     box._width, box._height, box._depth,
                                     simResult.BallPosList, box.getBallColors(), box.getBallRadii(), dT);

Visualizer::VisualizeParticleSimulation3D("air_as_ideal_gas.txt");
```

Using ParticleVisualizer3D

What air looks like in 100nm cube. Blue is nitrogen, red is oxygen, and if you try really hard, you might spot occasional yellow molecule representing Argon, but you'll need eagle's eye to spot CO2.

To be exact, in volume of 100nm cube, at STP, there are 20.967 molecules of N2, 5645 molecules of O2, 250 molecules of Ar, and 10 molecules of CO2.



PHYSICS OF IDEAL GAS

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

Little bit of formulas.

SIMULATING IDEAL GAS WITH OUR COLLISIONSIMULATOR

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”

Follow up projects and Exercises

Task 1 – more complex geometries of container

Task 2 – check Dalton law of partial pressure, modify recorder

Hole in the wall – how fast gas leaks

Continuous container –

Implement CollisionRecorder, verify mean free path formulas

5. Throwing things up in the air – everyday gravity

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
- Changing air resistance, depending on ball properties
- Effect of changing air density on cannon ball

From author perspective, it would be more logical to use soccer as an example, but alas, interaction between footballer's leg and soccer ball is, in its determination of initial conditions and simulation of motion, much, much complex problem, so baseball will do.

VACUUM SOLUTION

AIR RESISTANCE

AIR DENSITY EFFECT ON CANNON BALL

EFFECT OF VELOCITY ON BALL PROPERTIES AND BASEBALL DRAG

6. Pendulum – using our ODE solvers

Introducing forces … and putting to use emperors of numerical simulations, ODE solvers.

Tasks at hand:

- Introduce pendulum as one of the simplest physical systems
- Introduce numerical ODE solvers and solve exactly motion of pendulum
- Compare obtained solutions with analytical one
- Dumped and forced (driven) pendulum

PHYSICS OF PENDULUM

TODO – intro to pendulum

Applying Newton second law to tangential component, we get

$$F = -mg\sin\theta = ma,$$
$$a = -g\sin\theta, \qquad (5.1)$$

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:

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

Inserting this into equation 5.1, we get

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

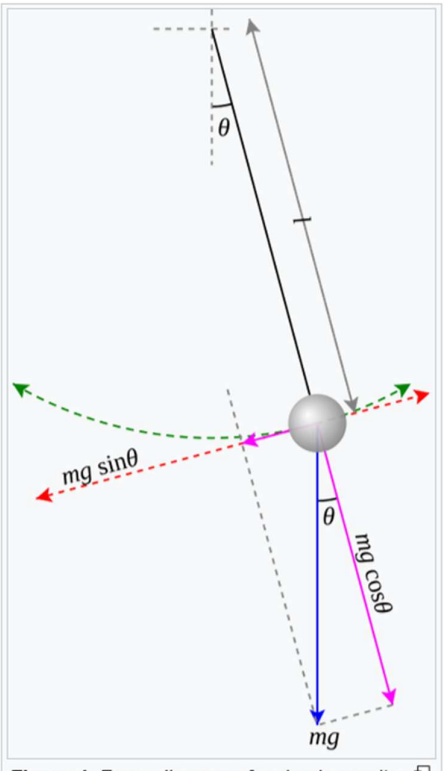


Figure 1. Force diagram of a simple gravity pendulum.

Picture from Wikipedia.

Giving us our final exact equation of motion for simple pendulum:

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

Exact solution involves elliptic integrals (more on that a little bit later), but we can employ linear approximation for small angles (θ << 1), where we approximate sin θ ≈ θ, which gives as an equation for harmonic oscillator, that is a linear differential equation of second order.

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

Solution for dependence of angle on time, if we assume initial angle θ_0 ,is given by:

$$\theta(t) = \theta_0 \cos\left(\sqrt{\frac{g}{\ell}}\, t\right) \qquad \theta_0 \ll 1.$$

With period of motion given by one of the most famous equations of high-school physics:

$$T_0 = 2\pi\sqrt{\frac{\ell}{g}}$$

PHYSICS OF HARMONIC OSCILLATOR

Damped harmonic oscillator

Forced harmonic oscillator

ANALYTICAL SOLUTION FOR EXACT PENDULUM EQUATION

How can we calculate exact values analytically?

Elliptic integrals come to play …

SOLVING PENDULUM NUMERICALLY

Starting with our differential equation of second order:

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

Transforming to a system of first-order differential equations

TODO

Which we can easily transcribe to 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]);
  }
};
```

For simple ODE systems as this, there is actually no need for creating a dedicated class, as ODE system can be completely defined with lambda function:

```
// alternative way of defining the system with lambda function
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]);
  }
);
```

As part of PendulumODE class definition, we'll add two functions for calculating period of the pendulum.

```
double calcPeriodLinearized()
{
  const double g = 9.81;
  double period = 2 * Constants::PI * std::sqrt(_length / g);
  return period;
}

double calcExactPeriod(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;
}
```

Solving pendulum with Euler method

We are starting with the simplest method.

```
// setting parameters for our simulation
Real  t1 = 0.0, t2 = 10.0;
int   expectNumSteps = 1000;              // means our step = 10 / 1000 = 0.001
Real  initAngle = Utils::DegToRad(30);    // initial angle set to 30 degrees
Vector<Real>  initCond{ initAngle, 0.0 };
```

```
// solving and getting solutions
ODESystemFixedStepSolver  fixedSolver(pendSys, StepCalculators::EulerStepCalc);
ODESystemSolution sol = fixedSolver.integrate(initCond, t1, t2, expectNumSteps);

Vector<Real> sol_x  = sol.getTValues();
Vector<Real> sol_y1 = sol.getXValues(0);
Vector<Real> sol_y2 = sol.getXValues(1);
```

To get console output we could do something like this

```
// console visualization
std::cout << "\n\n****  Euler method - fixed stepsize  **********\n";
std::vector<ColDesc>        vecNames{ ColDesc("t", 11, 2, 'F'),
                                      ColDesc("angle", 15, 8, 'F'),
                                      ColDesc("ang.vel.", 15, 8, 'F') };
std::vector<Vector<Real>*>  vecVals{ &sol_x, &sol_y1, &sol_y2 };

VerticalVectorPrinter vvp(vecNames, vecVals);

vvp.Print();
```

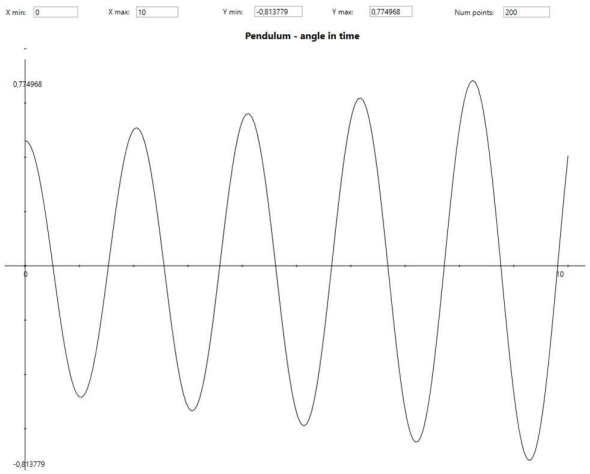
But we'll skip example until next section.

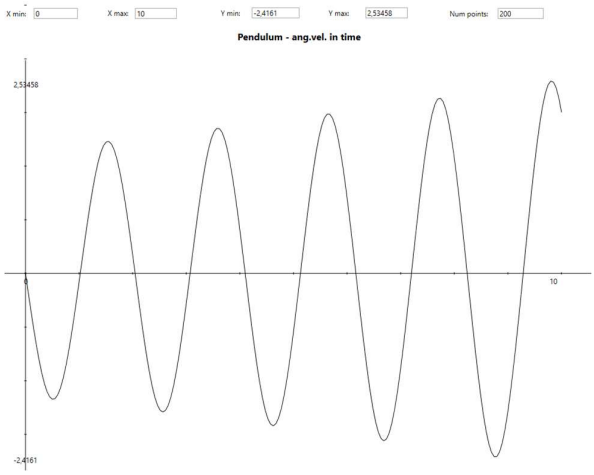
```
// visualizing solutions
PolynomInterpRealFunc   solInterpY1 = sol.getSolutionAsPolyInterp(0, 3);
PolynomInterpRealFunc   solInterpY2 = sol.getSolutionAsPolyInterp(1, 3);

Visualizer::VisualizeRealFunction(solInterpY1, "Pendulum - angle in time",
                                  t1, t2, 200, "pendulum_angle_euler.txt");

Visualizer::VisualizeRealFunction(solInterpY2, "Pendulum - ang.vel. in time",
                                  t1, t2, 200, "pendulum_ang_vel_euler.txt");
```

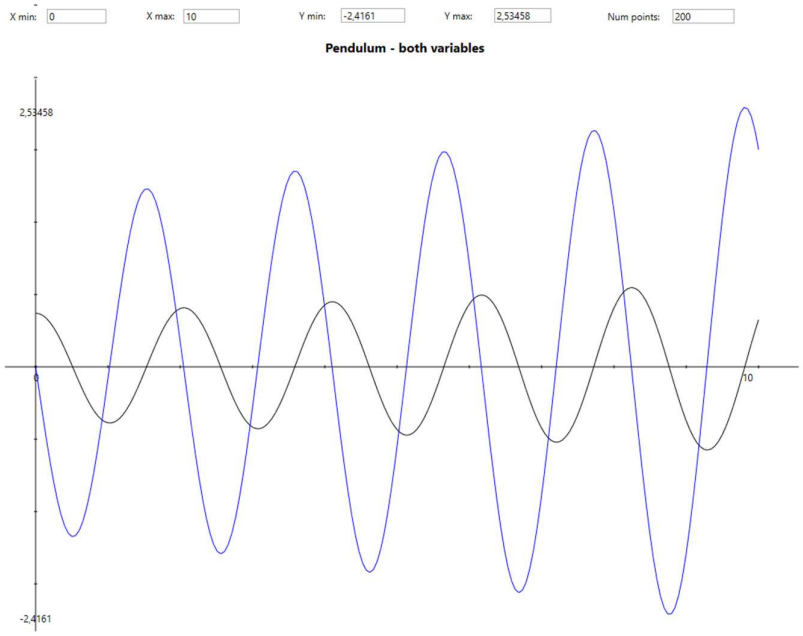
We can also visualize both variables (there are two ways, with the same result).

```
// shown together as multi-real function
Visualizer::VisualizeMultiRealFunction({ &solInterpY1, &solInterpY2 },
                                       "Pendulum - both variables", t1, t2, 200,
                                       "pendulum_multi_real_func_euler.txt");

// visualized together as ODE system solution
Visualizer::VisualizeODESysSolAsMultiFunc(sol, "Pendulum - Euler method",
                                          "pendulum_euler.txt");
```

Visualizations







Something is waaay off!?

NEED FOR SYMPLECTING INTEGRATORS THAT CONSERVE PHASE SPACE CONSTRAINTS.

TODO – mathematical analysis how Euler method discretization leads to ever increasing (kinetic) energy in the solutions.

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
   ...
   9.20  -0.49774494   0.13629050              9.38 -0.39991174  0.92399290
   9.30  -0.46106272   0.59202240              9.47 -0.29786175  1.24013268
```

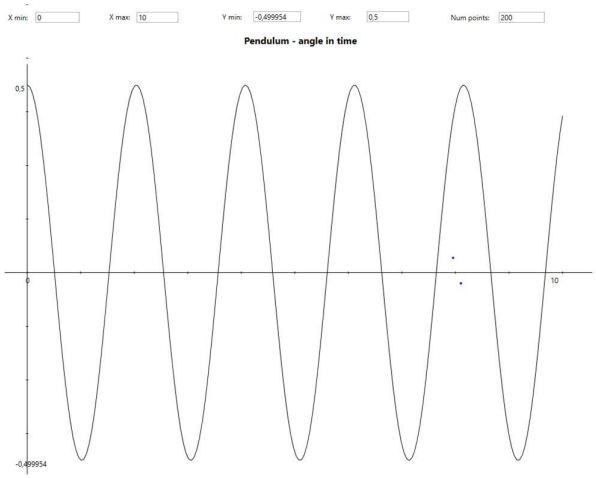
9.40	-0.38107371	0.99586669	9.57	-0.16254622	1.46398440
9.50	-0.26491163	1.30947278	9.68	-0.00443368	1.54973588
9.60	-0.12330802	1.50005315	9.83	0.22803362	1.37621501
9.70	0.03023798	1.54600550	9.92	0.33552554	1.14359632
9.80	0.18084654	1.44190607	10.00	0.41786505	0.84481904
9.90	0.31398298	1.19992327			
10.00	0.41709503	0.84673507			

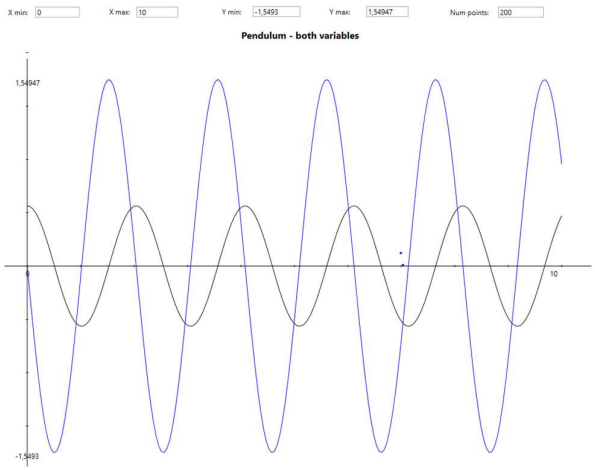
Visualizing solutions graphically

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.

We have a handy function in our FunctionsAnalyzer class, just for that.

```
Real calcRootsPeriod(Real t1, Real t2, int numPoints)
{
  Vector<Real> root_brack_x1(10), root_brack_x2(10);
  int numFoundRoots = FindRootBrackets(_f, t1, t2, numPoints, root_brack_x1, root_brack_x2);

  Vector<Real> roots(numFoundRoots);
  Vector<Real> rootDiffs(numFoundRoots - 1);
  for (int i = 0; i < numFoundRoots; i++)
  {
    roots[i] = FindRootBisection(_f, 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);
}
```

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;
}
```

TODO – add percentage difference between linear and exact?

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

ADDING AIR RESISTANCE – DAMPED PENDULUM

Basically, it is a damped oscillating system, but also nonlinear, in its real-world case.

```
class DumpedPendulumODE : public IODESystem
{
  Real _length, _dump_coeff;
public:
  DumpedPendulumODE(Real length, Real resistance)
       : _length(length), _dump_coeff(resistance) {}

  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]) - _dump_coeff * x[1];
  }
};
```

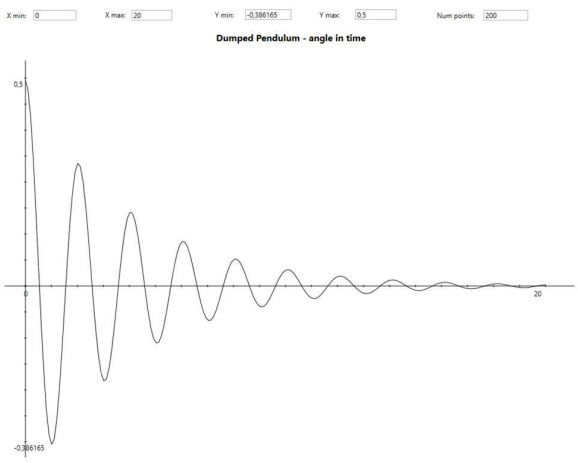
Our standard procedure for solving.

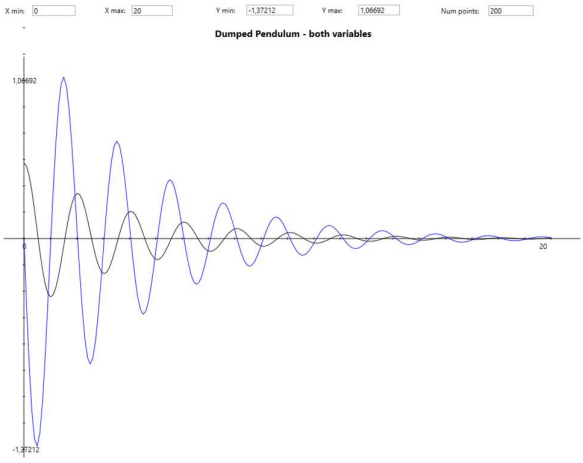
```
DumpedPendulumODE odeSys(1.0, 0.5);

Real  t1 = 0.0, t2 = 20.0;
int   expectNumSteps = 1000;
Real  initAngle = 0.5;
Vector<Real>  initCond{ initAngle, 0.0 };

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

Vector<Real> x_fixed = sol.getTValues();
Vector<Real> y1_fixed = sol.getXValues(0);
Vector<Real> y2_fixed = sol.getXValues(1);
```

And visualizing

```
PolynomInterpRealFunc   solInterpY1 = sol.getSolutionAsPolyInterp(0, 3);
PolynomInterpRealFunc   solInterpY2 = sol.getSolutionAsPolyInterp(1, 3);

Visualizer::VisualizeRealFunction(solInterpY1, "Dumped Pendulum - angle in time",
                                  t1, t2, 200, "dumped_pendulum_angle.txt");
// shown together
Visualizer::VisualizeMultiRealFunction({ &solInterpY1, &solInterpY2 },
                                       "Dumped Pendulum - both variables",
                                       t1, t2, 200,
                                       "dumped_pendulum_multi_real_func.txt");
```

Visualizations





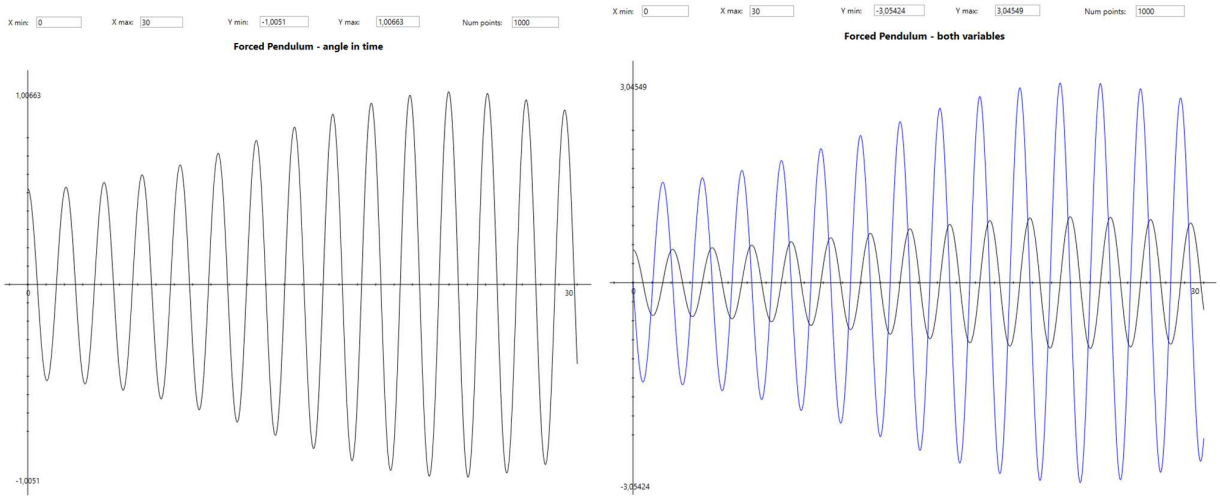
FORCED PENDULUM – RESONANCE

Interesting because of the appearance of resonance.

```
class ForcedPendulumODE : public IODESystem
{
  Real _l, _amp, _period;
public:
  ForcedPendulumODE(Real length, Real forceAmp, Real forcePeriod)
        : _l(length), _amp(forceAmp), _period(forcePeriod) {
  }
  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 / _l * sin(x[0]) + _amp * cos(Constants::PI * t / _period);
  }
};
```

Solving

Visualization



TODO – what happens with idealized linear system?

7. Spherical and double pendulum – Lagrangian enters the scene

Going all-in with Lagrangian formulation of classical mechanics.

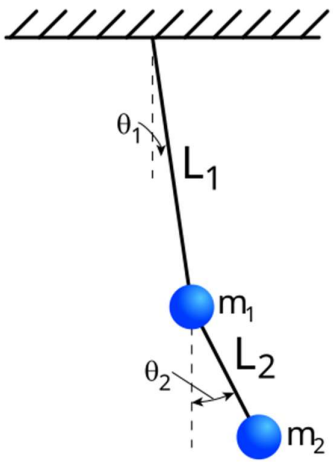
Tasks at hand:

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

LAGRANGIAN FORMULATION OF CLASSICAL MECHANICS

PHYSICS OF DOUBLE PENDULUM

There are various types of double pendulums – compound, …



Lagrangian for double pendulum:

$$L = \frac{1}{2}m_1(l_1\dot{\theta}_1)^2 + \frac{1}{2}m_2(l_1^2\dot{\theta}_1^2 + l_2^2\dot{\theta}_2^2 + 2l_1l_2\dot{\theta}_1\dot{\theta}_2 cos(\theta_1 - \theta_2))$$
$$+m_1gl_1cos(\theta_1) + m_2g(l_1cos(\theta_1) + l_2cos(\theta_2))$$

Solving Euler-Lagrange equations for given Lagrangian gives us two differential equations of motion:

$$(m_1 + m_2)l_1\ddot{\alpha}_1 + m_2l_2\ddot{\alpha}_2\cos(\alpha_1 - \alpha_2) + m_2l_2\dot{\alpha}_2^2\sin(\alpha_1 - \alpha_2) + (m_1 + m_2)g\sin\alpha_1 = 0$$
$$l_2\ddot{\alpha}_2 + l_1\ddot{\alpha}_1\cos(\alpha_1 - \alpha_2) - l_1\dot{\alpha}_1^2\sin(\alpha_1 - \alpha_2) + g\sin\alpha_2 = 0$$

Using computer algebra package, they can be solved to get expressions for theta1 and theta2

$$\theta_1'' = \frac{-g\,(2\,m_1 + m_2)\sin\theta_1 - m_2\,g\,\sin(\theta_1 - 2\,\theta_2) - 2\sin(\theta_1 - \theta_2)\,m_2\,(\theta_2'^2\,L_2 + \theta_1'^2\,L_1\cos(\theta_1 - \theta_2))}{L_1\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

$$\theta_2'' = \frac{2\sin(\theta_1 - \theta_2)\,(\theta_1'^2\,L_1\,(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + \theta_2'^2\,L_2\,m_2\cos(\theta_1 - \theta_2))}{L_2\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

Reducing to set of 4 first-order ODE's we get:

$$\theta_1' = \omega_1$$

$$\theta_2' = \omega_2$$

$$\omega_1' = \frac{-g\,(2\,m_1 + m_2)\sin\theta_1 - m_2\,g\,\sin(\theta_1 - 2\,\theta_2) - 2\sin(\theta_1 - \theta_2)\,m_2\,(\omega_2^2\,L_2 + \omega_1^2\,L_1\cos(\theta_1 - \theta_2))}{L_1\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

$$\omega_2' = \frac{2\sin(\theta_1 - \theta_2)\,(\omega_1^2\,L_1\,(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + \omega_2^2\,L_2\,m_2\cos(\theta_1 - \theta_2))}{L_2\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

No analytical solutions are known, so the only thing we can do is try to solve these equations numerically.

NUMERICALLY SOLVING DOUBLE PENDULUM

First, we define a class DoublePendulumODE, that will represent differential equations of motion.

```
class DoublePendulumODE : public IODESystem
{
  Real _m1, _m2;
  Real _l1, _l2;
public:
  DoublePendulumODE(Real m1, Real m2, Real l1, Real l2) : _m1(m1), _m2(m2), _l1(l1), _l2(l2) {}

  int  getDim() const override { return 4; }
  void derivs(const Real t, const MML::Vector<Real>& x, MML::Vector<Real>& dxdt) const override
  {
    Real th1 = x[0];
    Real w1  = x[1];
    Real th2 = x[2];
    Real w2  = x[3];

    Real g = 9.81;
    Real divisor = (2 * _m1 + _m2 - _m2 * cos(2*th1 - 2*th2));

    dxdt[0] = w1;
    dxdt[1] = (-g * (2 * _m1 + _m2) * sin(th1) - _m2 * g * sin(th1 - 2 * th2) -
               2 * sin(th1 - th2) * _m2 * (POW2(w2) * _l2 + POW2(w1) * _l1 * cos(th1 - th2))
              ) / (_l1 * divisor);
    dxdt[2] = w2;
    dxdt[3] = (2 * sin(th1 - th2) * (POW2(w1) * _l1 * (_m1 + _m2) +
               g * (_m1 + _m2) * cos(th1) +
               POW2(w2) * _l2 * _m2 * cos(th1 - th2))
              ) / (_l2 * divisor);
  }
};
```

Solving

```
void Demo_DoublePendulum()
{
  Real  l1 = 1.0, l2 = 1.0;
  Real  m1 = 0.5, m2 = 1.0;
  DoublePendulumODE   odeSysDoublePend = DoublePendulumODE(m1, m2, l1, l2);

  Real  t1 = 0.0, t2 = 10.0;
  int   expectNumSteps = 100;
  Real  minSaveInterval = (t2 - t1) / expectNumSteps;
  Real  initAngle1 = 0.5;
  Real  initAngle2 = 0.1;
  Vector<Real>  initCond{ initAngle1, 0.0, initAngle2, 0.0 };

  ODESystemSolver<RK4_CashKarp_Stepper> adaptSolver(odeSysDoublePend);
  ODESystemSolution sol = adaptSolver.integrate(initCond, t1, t2, minSaveInterval, 1e-06, 0.01);
```

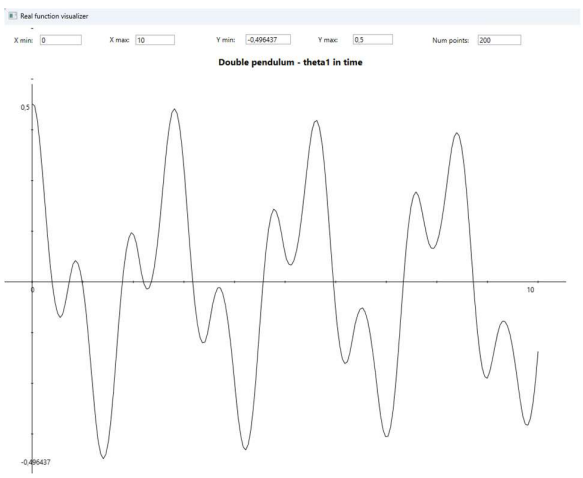
Printing in console

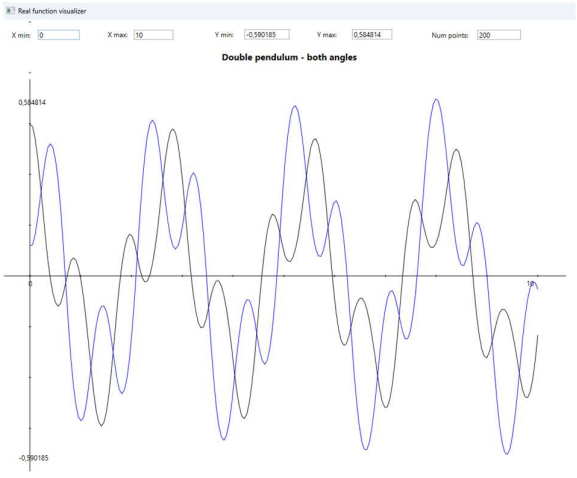
```
Vector<Real> t_vals      = sol.getXValues();
Vector<Real> theta1_vals = sol.getYValues(0);
Vector<Real> theta2_vals = sol.getYValues(2);

std::cout << "\n\n**** Solving double pendulum  ****\n";
std::vector<ColDesc>        vecNames{ ColDesc("t", 11, 2, 'F'),
                                      ColDesc("theta 1", 15, 8, 'F'),
                                      ColDesc("theta 2", 15, 8, 'F'), };
std::vector<Vector<Real>*>  vecVals{ &t_vals, &theta1_vals, &theta2_vals };

VerticalVectorPrinter vvp(vecNames, vecVals);

vvp.Print();
```

Visualizing graphically

```
// getting solutions as polynomials
PolynomInterpRealFunc   solAdaptPolyInterp0 = sol.getSolutionAsPolyInterp(0, 3);
PolynomInterpRealFunc   solAdaptPolyInterp1 = sol.getSolutionAsPolyInterp(2, 3);

Visualizer::VisualizeRealFunction(solAdaptPolyInterp0,
                                  "Double pendulum - theta1 in time",
                                  0.0, 10.0, 200, "double_pendulum_angle1.txt");

// shown together
Visualizer::VisualizeMultiRealFunction({ &solAdaptPolyInterp0, &solAdaptPolyInterp1 },
                                       "Double pendulum - both angles", 0.0, 10.0, 200,
                                       "double_pendulum_multi_real_func.txt");
```

Results



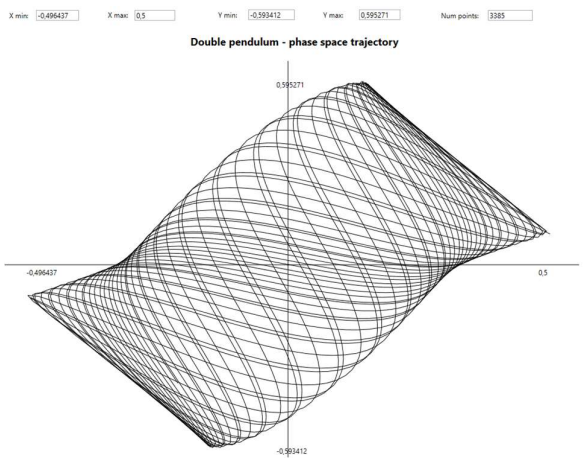


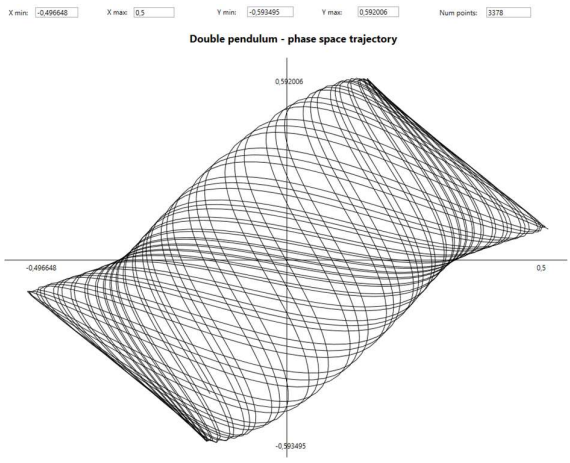
Visualizing phase space of two solutions with slightly different initial conditions.

We have to form a parametric curve in 2D from solutions.

```
// form parametric curve 2d from solution
Matrix<Real> curve_points(t_vals.size(), 2);
for (int i = 0; i < t_vals.size(); i++)
{
  curve_points(i, 0) = theta1_vals[i];
  curve_points(i, 1) = theta2_vals[i];
}

SplineInterpParametricCurve<2> phaseSpaceTrajectory(0.0, 1.0, curve_points);

Visualizer::VisualizeParamCurve2D(phaseSpaceTrajectory, "Double pendulum - phase space trajectory",
                                  0.0, 1.0, t_vals.size(), "double_pendulum_phase_space.txt");
```

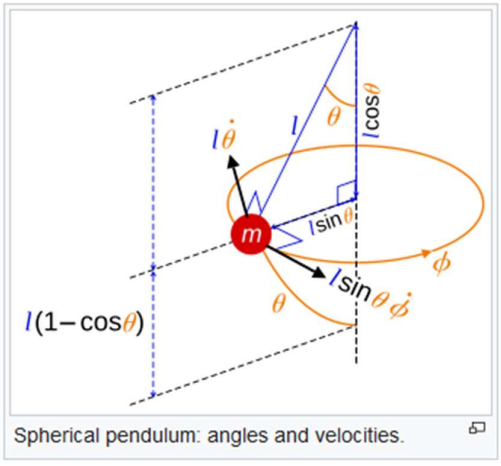
First : initAngle2 = 0.1 Second: initAngle2 = 0.101





PHYSICS OF SPHERICAL PENDULUM

Theoretical introduction mostly taken from Wikipedia (AND MUCH TO DO HERE!)



Spherical pendulum: angles and velocities.

Hamiltonian formulation.

The Hamiltonian is

$$H = P_\theta \dot{\theta} + P_\phi \dot{\phi} - L$$

where conjugate momenta are

$$P_\theta = \frac{\partial L}{\partial \dot{\theta}} = ml^2 \cdot \dot{\theta}$$

and

$$P_\phi = \frac{\partial L}{\partial \dot{\phi}} = ml^2 \sin^2\theta \cdot \dot{\phi}.$$

In terms of coordinates and momenta it reads

$$H = \underbrace{\left[\frac{1}{2}ml^2\dot{\theta}^2 + \frac{1}{2}ml^2 \sin^2\theta \dot{\phi}^2\right]}_{T} + \underbrace{\left[-mgl\cos\theta\right]}_{V} = \frac{P_\theta^2}{2ml^2} + \frac{P_\phi^2}{2ml^2 \sin^2\theta} - mgl\cos\theta$$

Where we are aiming for four first order differential equations of motion

$$\begin{aligned}
\dot{\theta} &= \frac{P_\theta}{ml^2} \\
\dot{\phi} &= \frac{P_\phi}{ml^2 \sin^2\theta} \\
\dot{P}_\theta &= \frac{P_\phi^2}{ml^2 \sin^3\theta}\cos\theta - mgl\sin\theta \\
\dot{P}_\phi &= 0
\end{aligned}$$

NUMERICALLY SOLVING SPHERICAL PENDULUM

We can represent this ODE's system as ODESystem class for our solvers.

```
class SphericalPendulumHamiltonODE : public IODESystem
{
  Real _m, _l;
public:
  SphericalPendulumHamiltonODE() : _l(1.0), _m(1.0) {}
  SphericalPendulumHamiltonODE(Real l) : _l(l), _m(1.0) {}
  SphericalPendulumHamiltonODE(Real m, Real l) : _m(m), _l(l) {}

  int  getDim() const override { return 4; }
  void derivs(const Real t, const MML::Vector<Real>& x, MML::Vector<Real>& dxdt) const override
  {
    Real tht   = x[0];
    Real phi   = x[1];
    Real P_tht = x[2];
    Real P_phi = x[3];

    Real g = 9.81;

    dxdt[0] = P_tht / (_m * POW2(_l));
    dxdt[1] = P_phi / (_m * POW2(_l * sin(tht)));
    dxdt[2] = POW2(P_phi) * cos(tht) / (_m * POW2(_l) * POW3(sin(tht))) - _m * g * _l * sin(tht);
    dxdt[3] = 0.0;
  }
};
```

When solving it, initial conditions are crucial.

```
SphericalPendulumHamiltonODE    odeSysSpherPend = SphericalPendulumHamiltonODE(l);

Real  t1 = 0.0, t2 = 20.0;
int   expectNumSteps = 2000;
Real  minSaveInterval = (t2 - t1) / expectNumSteps;

Real  initAngleTheta = Constants::PI / 2;
Real  initAnglePhi = 0.0;
Vector<Real>  initCond{ initAngleTheta, initAnglePhi, 5.10, -3.55 };
```

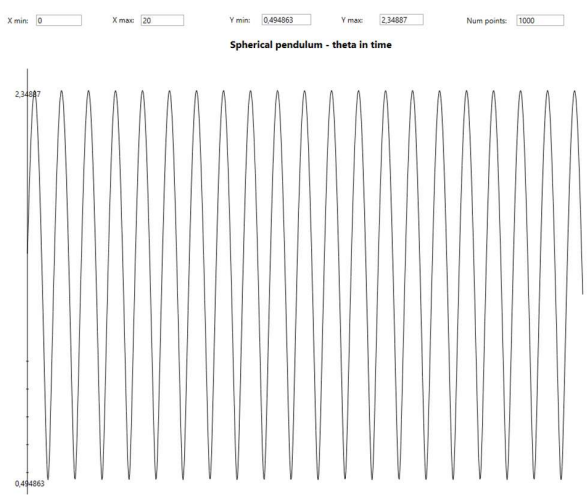
Solving it and getting solutions

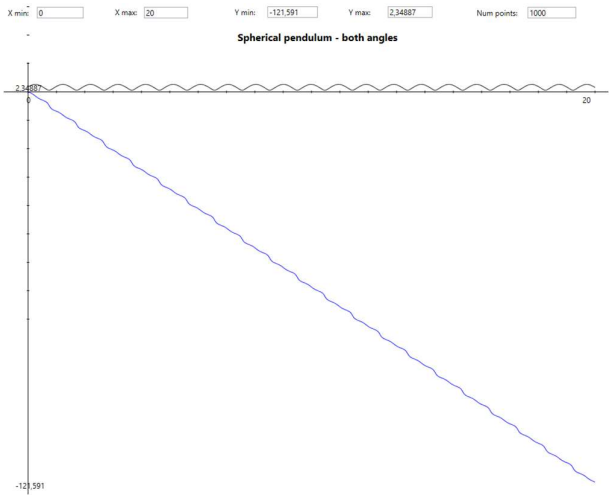
```
ODESystemSolver<RK5_CashKarp_Stepper> odeSolver(odeSysSpherPend);
ODESystemSolution odeSol = odeSolver.integrate(initCond, t1, t2, minSaveInterval, 1e-07, 0.01);

Vector<Real> t_vals     = odeSol.getTValues();
Vector<Real> theta_vals = odeSol.getXValues(0);
Vector<Real> phi_vals   = odeSol.getXValues(1);
Vector<Real> theta_dot_vals = odeSol.getXValues(2);
Vector<Real> phi_dot_vals   = odeSol.getXValues(3);
```

Visualizing solutions

```
// getting solutions as polynomials
PolynomInterpRealFunc   solAdaptPolyInterp0 = odeSol.getSolutionAsPolyInterp(0, 3);
PolynomInterpRealFunc   solAdaptPolyInterp1 = odeSol.getSolutionAsPolyInterp(1, 3);

Visualizer::VisualizeRealFunction(solAdaptPolyInterp0, "Spherical pendulum - theta in time",
                                  t1, t2, 200, "spherical_pendulum_theta.txt");

// shown together
Visualizer::VisualizeMultiRealFunction({ &solAdaptPolyInterp0, &solAdaptPolyInterp1 },
                                       "Spherical pendulum - both angles", { "Theta", "Phi" },
                                       t1, t2, 200,
                                       "spherical_pendulum_multi_real_func.txt");
```

OUPS!!!





There's a surprise!!! TODO – investigate! 😍

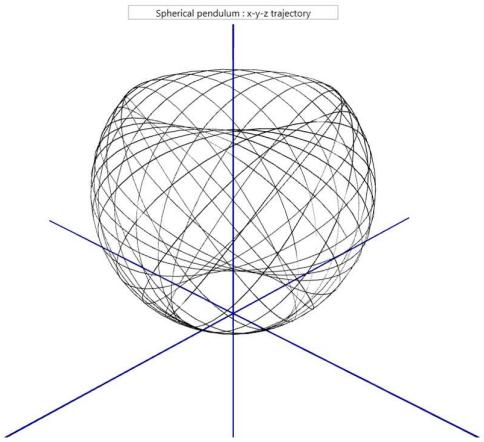
Visualization of phase space

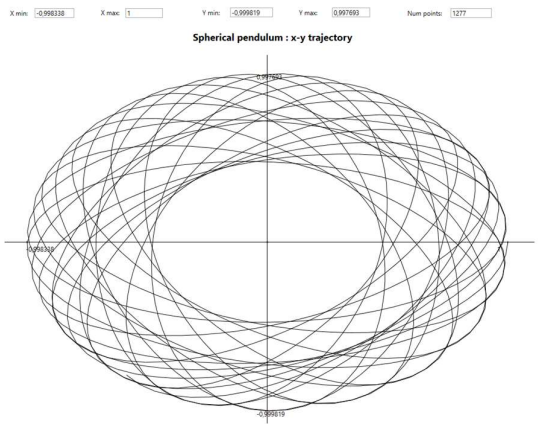
```
// projecting pendulum path on x-y, x-z and y-z plane
// first, get values for x, y, z coordinates in vectors
Vector<Real> x_vals(t_vals.size()), y_vals(t_vals.size()), z_vals(t_vals.size());

for (int i = 0; i < t_vals.size(); i++)
{
  x_vals[i] = l * sin(theta_vals[i]) * cos(phi_vals[i]);
  y_vals[i] = l * sin(theta_vals[i]) * sin(phi_vals[i]);
  z_vals[i] = l * (1 - cos(theta_vals[i]));
}
```

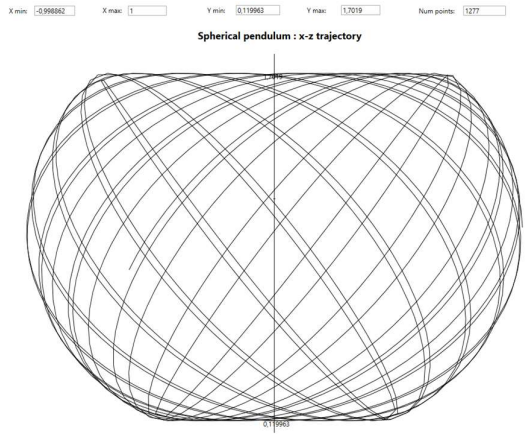
```
// then form appropriate matrices with relevant data for visualization of parametric curves
Matrix<Real> curve_x_y_points(t_vals.size(), 2), curve_x_z_points(t_vals.size(), 2);
Matrix<Real> curve_y_z_points(t_vals.size(), 2);
Matrix<Real> curve_xyz_points(t_vals.size(), 3);
for (int i = 0; i < t_vals.size(); i++)
{
  curve_xyz_points(i, 0) = 100 * x_vals[i];
  curve_xyz_points(i, 1) = 100 * y_vals[i];
  curve_xyz_points(i, 2) = 100 * z_vals[i];

  curve_x_y_points(i, 0) = x_vals[i];
  curve_x_y_points(i, 1) = y_vals[i];

  curve_x_z_points(i, 0) = x_vals[i];
  curve_x_z_points(i, 1) = z_vals[i];

  curve_y_z_points(i, 0) = y_vals[i];
  curve_y_z_points(i, 1) = z_vals[i];
}
```

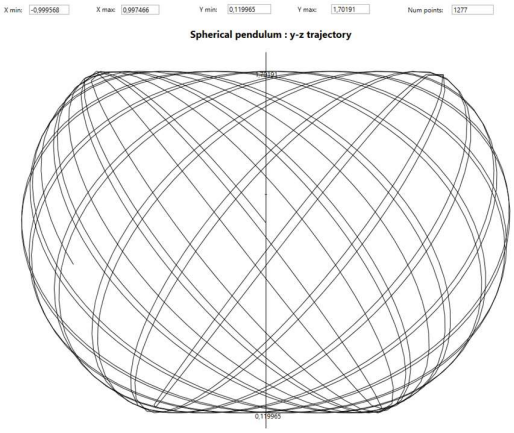
Visualizing

```
SplineInterpParametricCurve<3> xyzTrajectory(0.0, 1.0, curve_xyz_points);
Visualizer::VisualizeParamCurve3D(xyzTrajectory, "Spherical pendulum : x-y-z trajectory",
                                  0.0, 1.0, t_vals.size(), "spherical_pendulum_xyz_trajectory.txt");

SplineInterpParametricCurve<2> xyTrajectory(0.0, 1.0, curve_x_y_points);
Visualizer::VisualizeParamCurve2D(xyTrajectory, "Spherical pendulum : x-y trajectory",
                                  0.0, 1.0, t_vals.size(), "spherical_pendulum_xy_trajectory.txt");

SplineInterpParametricCurve<2> xzTrajectory(0.0, 1.0, curve_x_z_points);
Visualizer::VisualizeParamCurve2D(xzTrajectory, "Spherical pendulum : x-z trajectory",
                                  0.0, 1.0, t_vals.size(), "spherical_pendulum_xz_trajectory.txt");

SplineInterpParametricCurve<2> yzTrajectory(0.0, 1.0, curve_y_z_points);
Visualizer::VisualizeParamCurve2D(yzTrajectory, "Spherical pendulum : y-z trajectory",
                                  0.0, 1.0, t_vals.size(), "spherical_pendulum_yz_trajectory.txt");
```


Spherical pendulum : x-y-z trajectory


X min: -0.998338
X max: 1
Y min: -0.999819
Y max: 0.997693
Num points: 1277
Spherical pendulum : x-y trajectory
0.997693
-0.998338
-0.999819


X min: -0.998862
X max: 1
Y min: 0.119963
Y max: 1.7019
Num points: 1277
Spherical pendulum : x-z trajectory
1.7019
0.119963


X min: -0.999568
X max: 0.997466
Y min: 0.119965
Y max: 1.70191
Num points: 1277
Spherical pendulum : y-z trajectory
1.70191
0.119965

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.
- Gravitational slingshot

PHYSICS OF GRAVITATIONAL FIELD

Basic formulas

Kepler laws

SOLVING TWO BODY PROBLEM NUMERICALLY

As two body problem is analytically completely solvable, it is unnecessary to employ numerical methods for obtaining solutions. However, these implementations will be foundation for N-body simulator.

Creating TwoBodySimulator2D class

Starting with 2D case

Class modeling state of the (gravitational) body. Radius and color are both exclusively “visualization” properties, and are not used in any calculations.

```
class GravityBodyState2D
{
private:
  Real  _mass;
  Real  _radius;
  std::string _color;

  Vec2Cart  _position;
  Vec2Cart  _velocity;
```

And its public interface

```
public:
  GravityBodyState2D() { ... }
  GravityBodyState2D(const Real& mass, const Vec2Cart& position) { ... }
  GravityBodyState2D(const Real& mass, const Vec2Cart& position, const Vec2Cart& velocity)
  GravityBodyState2D(const Real& mass, const Vec2Cart& position, const Vec2Cart& velocity,
                     const Real& radius, const std::string& color) { ... }

  Real  Mass()  const { return _mass; }
  Real& Mass()        { return _mass; }
  Real Radius() const { return _radius; }
  Real& Radius()      { return _radius; }
  std::string Color() const { return _color; }
  std::string& Color()      { return _color; }

  Vec2Cart  R() const { return _position; }
  Vec2Cart& R()       { return _position; }
  Vec2Cart  V() const { return _velocity; }
  Vec2Cart& V()       { return _velocity; }
};
```

Modeling state of two body system

Members, constructors and accessors

```
class TwoBodyState2D
{
private:
  GravityBodyState2D _body1;
  GravityBodyState2D _body2;

public:
  TwoBodyState2D() { }
  TwoBodyState2D(const GravityBodyState2D& m1, const GravityBodyState2D& m2) { ... }

  const GravityBodyState2D& Body1() const { return _body1; }
  const GravityBodyState2D& Body2() const { return _body2; }

  GravityBodyState2D& BodyAcc1() { return _body1; }
  GravityBodyState2D& BodyAcc2() { return _body2; }

  Vec2Cart Pos1() const { return _body1.R(); }
  Vec2Cart Pos2() const { return _body2.R(); }
  Vec2Cart V1()   const { return _body1.V(); }
  Vec2Cart V2()   const { return _body2.V(); }
```

Calculators of different properties

```
Real Dist() const { ... }

Vec2Cart CenterOfMassPos() const { ... }
Vec2Cart CenterOfMassVelocity() const { ... }

Real ReducedMass() const { return _body1.Mass() * _body2.Mass() / (_body1.Mass() + _body2.Mass()); }

Real KineticEnergy1()     const { return 0.5 * _body1.Mass() * POW2(_body1.V().NormL2()); }
Real KineticEnergy2()     const { return 0.5 * _body2.Mass() * POW2(_body2.V().NormL2()); }
Real TotalKineticEnergy() const { return KineticEnergy1() + KineticEnergy2(); }

Real TotalPotentialEnergy(Real G) const { ... }
```

Next is class representing initial configuration state, and also containing basic simulation parameter, value of gravity constant G.

```
class TwoBodyGravitySimConfig2D
{
public:
  Real _G = 100;
  TwoBodyState2D _initState;

  TwoBodyGravitySimConfig2D(const GravityBodyState2D& m1, const GravityBodyState2D& m2) { ... }
  TwoBodyGravitySimConfig2D(const Real& G, const GravityBodyState2D& m1, const GravityBodyState2D& m2)

  Real Mass1() const { return _initState.Body1().Mass(); }
  Real Mass2() const { return _initState.Body2().Mass(); }

  Vector<Real> getInitCond()
  {
    return Vector<Real>{_initState.Pos1()[0], _initState.Pos1()[1],
                        _initState.Pos2()[0], _initState.Pos2()[1],
                        _initState.V1()[0], _initState.V1()[1],
                        _initState.V2()[0], _initState.V2()[1] };
  }
};
```

Important thing – we first have position coordinates for all the bodies, and after that come velocity values.

That convention is also visible in class representing ODE system

```
class TwoBodyGravitySystemODE2D : public IODESystem
{
  TwoBodyGravitySimConfig2D _config;

public:
  TwoBodyGravitySystemODE2D(const TwoBodyGravitySimConfig2D& config) { ... }

  int getDim() const override { return 8; }
  void derivs(const Real t, const Vector<Real>& x, Vector<Real>& dxdt) const override
  {
    Vec2Cart r1{ x[0], x[1] }, r2{ x[2], x[3] };
    Vec2Cart v1{ x[4], x[5] }, v2{ x[6], x[7] };

    Vec2Cart r12 = r2 - r1;
    Real dist = r12.NormL2();

    Vec2Cart f12 = _config._G * _config.Mass1() * _config.Mass2() / POW3(dist) * r12;
    Vec2Cart f21 = -f12;

    dxdt[0] = v1[0];
    dxdt[1] = v1[1];
    dxdt[2] = v2[0];
    dxdt[3] = v2[1];
    dxdt[4] = f12[0] / _config.Mass1();
    dxdt[5] = f12[1] / _config.Mass1();
    dxdt[6] = f21[0] / _config.Mass2();
    dxdt[7] = f21[1] / _config.Mass2();
  }
};
```

We also need a class for handling simulation results.

```
class TwoBodyGravitySimulationResults2D
{
public:
  const TwoBodyGravitySimConfig2D& _config;

  Real _duration;
  Vector<TwoBodyState2D> _vecStates;
  Vector<Real> _vecTimes;

  TwoBodyGravitySimulationResults2D(const TwoBodyGravitySimConfig2D& config) { ... }
  TwoBodyGravitySimulationResults2D(const TwoBodyGravitySimConfig2D& config, const Real& duration)
```

Basic accessors

```
int   NumSteps() const { return (int)_vecStates.size(); }
Real  Duration() const { return _duration; }

Real            Time(int i)  const { return _vecTimes[i]; }
TwoBodyState2D  State(int i) const { return _vecStates[i]; }

Vector<Real>     getTimes() const { return _vecTimes; }
Vector<Vec2Cart> getBody1Pos() const { ... }
Vector<Vec2Cart> getBody2Pos() const { ... }
Vector<Vec2Cart> getBody1Velocity() const { ... }
Vector<Vec2Cart> getBody2Velocity() const { ... }
```

Extended set of utilities

```
Vector<Real> getPos1X() const { ... }
Vector<Real> getPos1Y() const { ... }
Vector<Real> getPos2X() const { ... }
Vector<Real> getPos2Y() const { ... }

Vector<Real> getV1X() const { ... }
Vector<Real> getV1Y() const { ... }
Vector<Real> getV2X() const { ... }
Vector<Real> getV2Y() const { ... }

Vector<Real> getCMPosX() const { ... }
Vector<Real> getCMPosY() const { ... }

Vector<Real> getCMVX() const { ... }
Vector<Real> getCMVY() const { ... }

Vector<Real> getTotalKineticEnergy() const { ... }
Vector<Real> getTotalPotentialEnergy() const { ... }
Vector<Real> getTotalEnergy() const { ... }

// visualize as parametric curve
void VisualizeSolution(std::string baseFileName) { ... }
```

And finally, simulator class, that is, as of now, quite simple.

```
class TwoBodiesGravitySimulator2D
{
  TwoBodyGravitySimConfig2D _config;

public:
  TwoBodiesGravitySimulator2D(const TwoBodyGravitySimConfig2D& config) { ... }

  TwoBodyGravitySimulationResults2D SolveRK5(Real duration, Real eps, Real minSaveInterval, Real hStart)
  TwoBodyGravitySimulationResults2D SolveRK5(Real duration, Real eps, Real minSaveInterval) { ... }
  TwoBodyGravitySimulationResults2D SolveRK5(Real duration, Real eps) { ... }
  TwoBodyGravitySimulationResults2D SolveRK5(Real duration) { ... }

  // get trajectory type - Ellipse, Parabolic, Hyperbolic
  std::string GetTrajectoryType(const TwoBodyGravitySimConfig2D &sysConfig) { ... }

  // solve and visualize trajectory
  void SolveAndShowTrajectories(Real t) { ... }
};
```

Testing TwoBodySimulator2D

Config generator comes to play, where we have extensive set of configurations for different cases

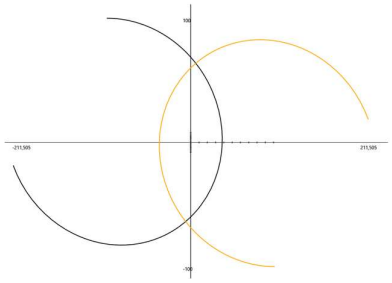
```
class TwoBodyGravityConfigGenerator2D
{
public:
  // config for two same bodies, in elliptic orbit, center of mass static
  static TwoBodyGravitySimConfig2D Config1_same_bodies_elliptic_CM_static(Real G = 1) { ... }

  // config for two same bodies, in elliptic orbit, center of mass moving
  static TwoBodyGravitySimConfig2D Config2_same_bodies_elliptic_CM_moving(Real G = 1) { ... }

  // config for two different bodies, in elliptic orbit, center of mass static
  static TwoBodyGravitySimConfig2D Config3_diff_bodies_elliptic_CM_static(Real G = 1) { ... }

  // config for two different bodies, in elliptic orbit, center of mass moving
  static TwoBodyGravitySimConfig2D Config4_diff_bodies_elliptic_CM_moving(Real G = 1) { ... }

  // config for two same bodies, in hyperbolic orbit, center of mass static+
  static TwoBodyGravitySimConfig2D Config5_same_bodies_hyperbolic_CM_static(Real G = 1) { ... }

  // config for two same bodies, in hyperbolic orbit, center of mass static+
  static TwoBodyGravitySimConfig2D Config6_same_bodies_hyperbolic_CM_moving(Real G = 1) { ... }
};
```

Testing setup

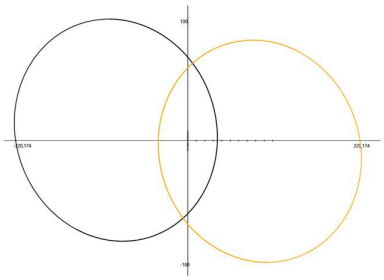
```
TwoBodyGravitySimConfig2D config = TwoBodyGravityConfigGenerator2D::Config1_same_bodies_elliptic_CM_static();
//TwoBodyGravitySimConfig2D config = TwoBodyGravityConfigGenerator2D::Config2_same_bodies_elliptic_CM_moving();
//TwoBodyGravitySimConfig2D config = TwoBodyGravityConfigGenerator2D::Config3_diff_bodies_elliptic_CM_static();
//TwoBodyGravitySimConfig2D config = TwoBodyGravityConfigGenerator2D::Config4_diff_bodies_elliptic_CM_moving();
//TwoBodyGravitySimConfig2D config = TwoBodyGravityConfigGenerator2D::Config5_same_bodies_hyperbolic_CM_static();
//TwoBodyGravitySimConfig2D config = TwoBodyGravityConfigGenerator2D::Config6_same_bodies_hyperbolic_CM_moving();

TwoBodiesGravitySimulator2D sim(config);

auto result = sim.SolveRK5(100, 1e-08, 0.5);
//auto result = sim.SolveRK5(300, 1e-08, 0.5);
//auto result = sim.SolveRK5(500, 1e-08, 0.5);
//auto result = sim.SolveRK5(1000, 1e-08, 0.5);

result.VisualizeSolution("gravity_example_2d");
```

Elliptic orbit, same bodies, CM static

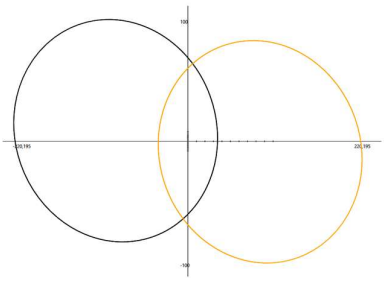
Num steps = 100



Num steps = 200

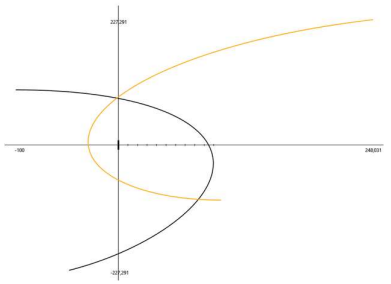


Num steps = 500

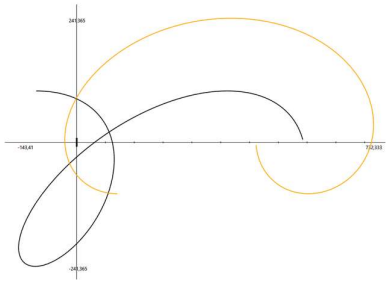


Elliptic orbit, same bodies, CM moving

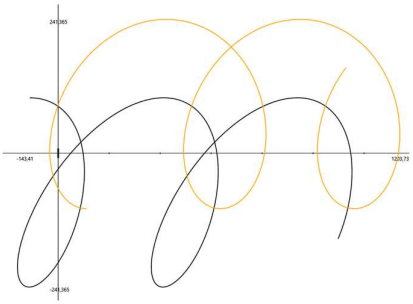
N = 100



N = 500

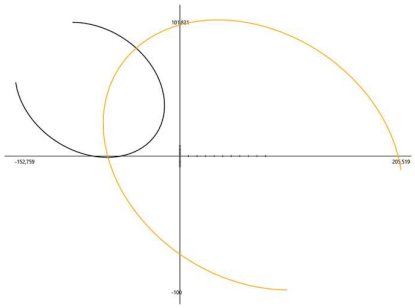


N = 1000

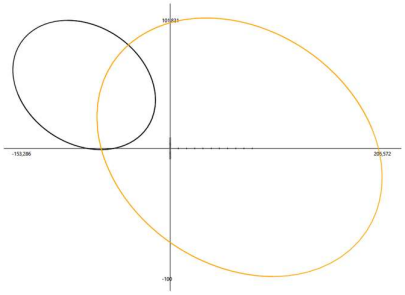


Elliptic orbit, different bodies (m1 = 2 * m2), CM static

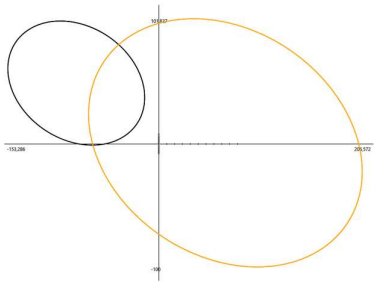
Num steps = 100



Num steps = 200

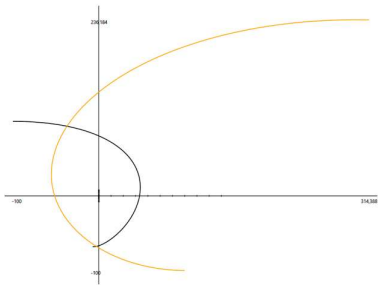


Num steps = 500

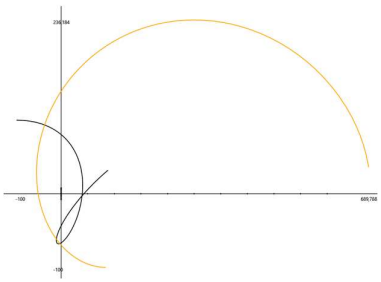


Elliptic orbit, different bodies (m1 = 2 * m2), CM moving

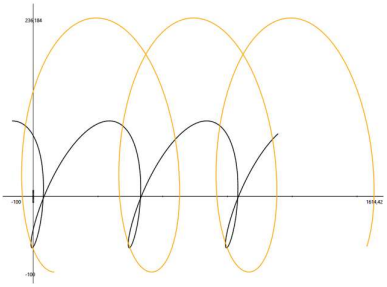
N = 100



N = 500

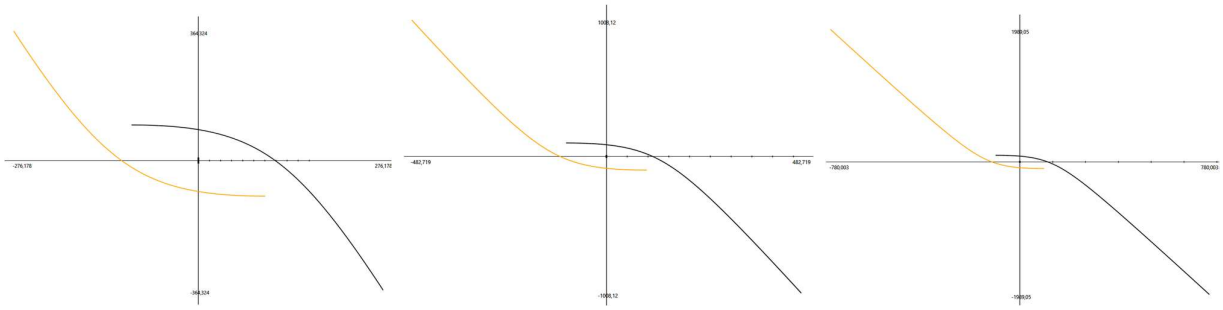


N = 1000



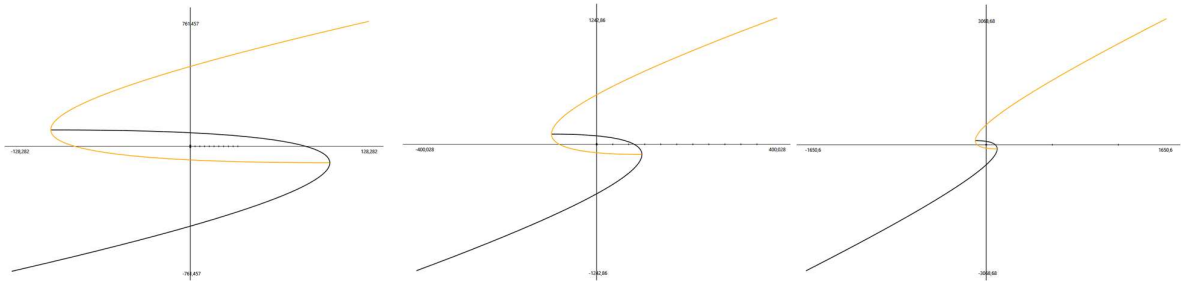
Example with hyperbolic orbit, same bodies m = 10000, v0 = 7 and -7.

Num steps = 100 Num steps = 200 Num steps = 500



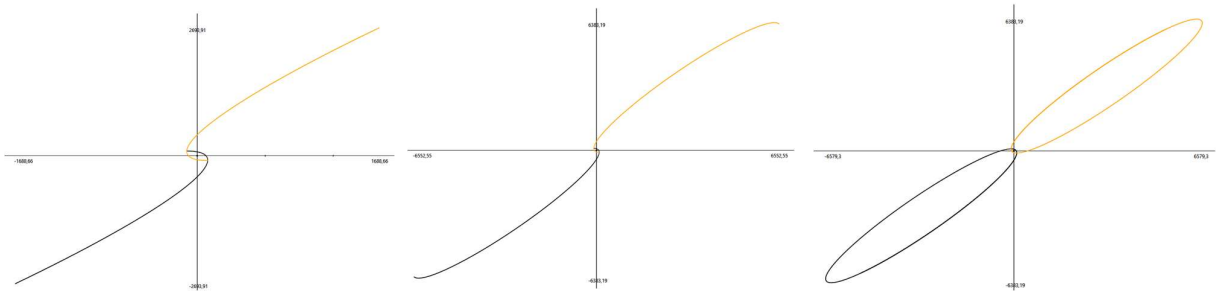
If we get v to almost critical value of 5,95, we still get hyperbolic trajectories.

Num steps = 250 Num steps = 500 Num steps = 500



But if we lower velocity to 5,9, we get elliptical trajectory, even if we need many, many steps to see the ellipse.

N = 2000 N = 20.000 N = 50.000



Creating and using TwoBodySimulator3D class

Extending our code for 2D case to 3D case is completely trivial, with just one addition – in 3D case we can also observe and calculate angular momentum, so a little bit of extension for simulation results class.

And our 3D simulator is much the same as 2D.

```
class TwoBodiesGravitySimulator
{
  TwoBodyGravitySimConfig _config;

public:
  TwoBodiesGravitySimulator(const TwoBodyGravitySimConfig& config) { ... }

  TwoBodyGravitySimulationResults SolveRK5(Real duration, Real eps, Real minSaveInterval, Real hStart) { ... }
  TwoBodyGravitySimulationResults SolveRK5(Real duration, Real eps, Real minSaveInterval) { ... }
  TwoBodyGravitySimulationResults SolveRK5(Real duration, Real eps) { ... }
  TwoBodyGravitySimulationResults SolveRK5(Real duration) { ... }

  // get trajectory type - Ellipse, Parabolic, Hyperbolic
  std::string GetTrajectoryType(const TwoBodyGravitySimConfig &sysConfig) { ... }

  // get plane of motion
  Plane3D GetPlaneOfMotion(const TwoBodyGravitySimConfig &sysConfig) { ... }

  // solve and visualize trajectory
  void SolveAndShowTrajectories(Real t) { ... }
};
```

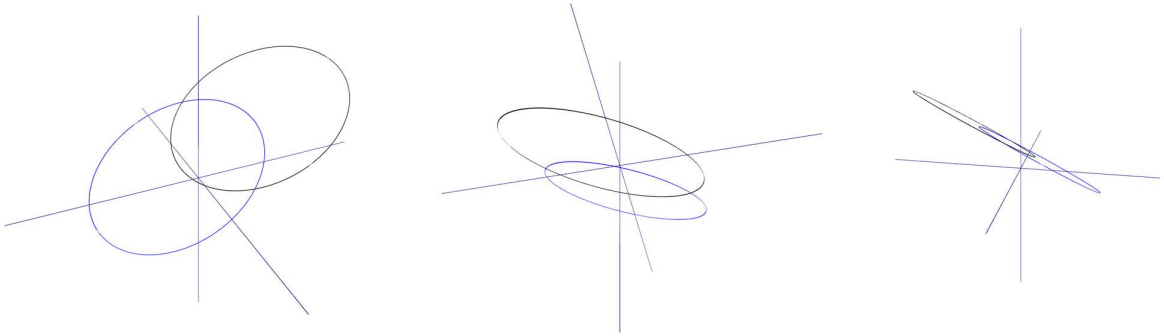
Of course, we also need to use 3D versions of our visualizers, and we will be using two different configurations from our TwoBodyConfigGenerator class

```
static TwoBodyGravitySimConfig Config1_same_bodies_elliptic_CM_static()
{
  Real G = 1;
  GravityBodyState m1(10000, Vec3Cart{ -100, 100, 50 }, Vec3Cart{ 4, 0, -2 });
  GravityBodyState m2(10000, Vec3Cart{ 50, -100, 50 }, Vec3Cart{ -4, 0, 2 });

  return TwoBodyGravitySimConfig(G, m1, m2);
}
```

And.

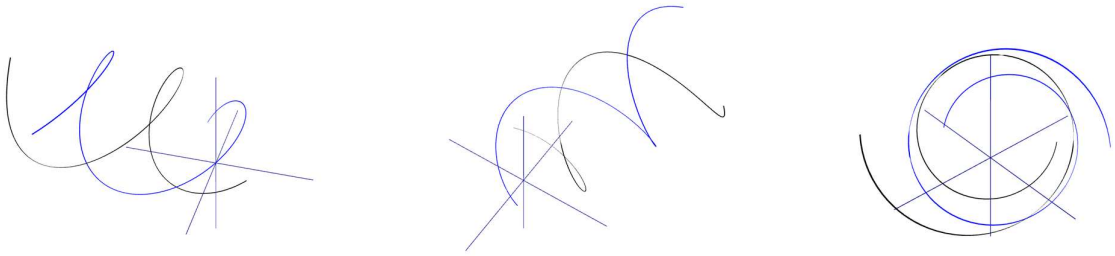
```
static TwoBodyGravitySimConfig Config2_same_bodies_elliptic_CM_moving()
{
  Real G = 1;
  GravityBodyState m1(10000, Vec3Cart{ -100, 100, 50 }, Vec3Cart{ 4, 3, -2 });
  GravityBodyState m2(10000, Vec3Cart{ 50, -100, 50 }, Vec3Cart{ -2, -1, 4 });

  return TwoBodyGravitySimConfig(G, m1, m2);
}
```

In the first case, center of mass is not moving, and we get two nice ellipses in 3D.



Where from the last picture is obvious that bodies are actually moving in a plane.

In the second case, with the center of mass moving, there is no single plane of motion.



TODO – project to center-of-mass frame and show ellipses again!

INTRODUCING FIELD OPERATIONS – GRAD, DIV, CURLS AND ALL THAT JAZZ

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

Contour plot of gravity potential as basis for gradient definition

Gradient

Divergence

Curl

Laplacian

IMPLEMENTING FIELD OPERATIONS IN C++

Scalar field operations – gradient and Laplacian

First, general version of gradient, with given metric tensor.

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

Gradient calculation in different coordinate systems.

```
    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) { ... }
```

Laplacian calculation in different coordinate systems.

```
    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) { ... }
```

Example – implementation of 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

Set of operations for calculating divergence and curl in different coordinate systems.

```
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

```cpp
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;
}
```

```cpp
static Vec3Cyl  CurlCyl(const IVectorFunction<3>& vectorField, const VectorN<Real, 3>& pos)
{
  VectorN<Real, 3> vals = vectorField(pos);

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

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

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

  Vector3Cylindrical ret{(1 / pos[0] * dzdphi - dphidz), drdz - dzdr, 1 / pos[0] * (vals[1] + pos[0] * dphidr - drdphi)};

  return ret;
}
```

```cpp
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;
}
```

Testing implementation on gravity field

For gravitational field we can calculate analytically values for gradient, divergence and curl, and it is a perfect situation to verify how good are our numerically calculated values.

MATH - PATH INTEGRATION

IMPLEMENTING PATH INTEGRATION IN C++

We'll create a separate class to hold path integral algorithms implementations.

```
class PathIntegration
{
  template<int N>
  class HelperCurveLen { ... };

  template<int N>
  class HelperCurveMass { ... };

  template<int N>
  class HelperLineIntegralScalarFunc { ... };

  template<int N>
  class HelperLineIntegralVectorFunc { ... };

public:
  template<int N>
  static Real ParametricCurveLength(const IParametricCurve<N>& curve, const Real a, const Real b) { ... }

  template<int N>
  static Real ParametricCurveMass(const IParametricCurve<N>& curve, const IRealFunction &density,
                                  const Real a, const Real b) { ... }

  static Real LineIntegral(const IScalarFunction<3>& scalarField, const IParametricCurve<3>& curve,
                           const Real a, const Real b, const Real eps = Defaults::WorkIntegralPrecision) { ... }

  static Real LineIntegral(const IVectorFunction<3>& vectorField, const IParametricCurve<3>& curve,
                           const Real a, const Real b, const Real eps = Defaults::LineIntegralPrecision) { ... }
};
```

We have two important utility classes that are actually defining real functions to integrate, based on provided fields and path, represented by ParametricCurve

```
template<int N>
class HelperLineIntegralScalarFunc : public IRealFunction
{
  const IScalarFunction<N>& _scalar_field;
  const IParametricCurve<N>& _curve;
public:
  HelperLineIntegralScalarFunc(const IScalarFunction<N>& scalarField, const IParametricCurve<N>& curve)
      : _scalar_field(scalarField), _curve(curve) {}

  Real operator()(Real t) const
  {
    auto tangent_vec = Derivation::DeriveCurve<N>(_curve, t, nullptr);

    auto field_val = _scalar_field(_curve(t));
    auto ret = field_val * tangent_vec.NormL2();

    return ret;
  }
};
```

And the second one is used when we have a vector field

```
template<int N>
class HelperLineIntegralVectorFunc : public IRealFunction
{
  const IVectorFunction<N>& _vector_field;
  const IParametricCurve<N>& _curve;
public:
  HelperLineIntegralVectorFunc(const IVectorFunction<N>& vectorField, const IParametricCurve<N>& curve)
      : _vector_field(vectorField), _curve(curve) {}

  Real operator()(Real t) const
  {
    auto tangent_vec = Derivation::DeriveCurve<N>(_curve, t, nullptr);
    auto field_vec = _vector_field(_curve(t));

    return Utils::ScalarProduct<N>(tangent_vec, field_vec);
  }
};
```

Implementation of functions that will be doing concrete integration is trivial:

```
static Real LineIntegral(const IScalarFunction<3>& scalarField, const IParametricCurve<3>& curve,
                         const Real t1, const Real t2, const Real eps = Defaults::WorkIntegralPrecision)
{
  HelperLineIntegralScalarFunc helper(scalarField, curve);

  return IntegrateTrap(helper, t1, t2, nullptr, nullptr, eps);
}

static Real LineIntegral(const IVectorFunction<3>& vectorField, const IParametricCurve<3>& curve,
                         const Real t1, const Real t2, const Real eps = Defaults::LineIntegralPrecision)
{
  HelperLineIntegralVectorFunc helper(vectorField, curve);

  return IntegrateTrap(helper, t1, t2, nullptr, nullptr, eps);
}
```

Testing implementation on gravity field

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

Setup:

```
Real G = 1.0;
Real mass = 100.0;

GravityPotentialField   potentialField(G, mass, Vec3Cart(0, 0, 0));
GravityForceField       forceField(G, mass, Vec3Cart(0, 0, 0));

// our two points in space
Vec3Cart point1(10.0, 10.0, -5.0);
Vec3Cart point2(-5.0, 5.0, 8.0);
```

Calculating potential :

```
Real potential1 = potentialField(point1);
Real potential2 = potentialField(point2);

std::cout << "Potential at point 1: " << potential1 << std::endl;
std::cout << "Potential at point 2: " << potential2 << std::endl;
std::cout << "Difference in potential: " << potential1 - potential2 << std::endl;
```

Potential difference based on path integral calculation, along a simple line connecting two points:

```
Curves::LineCurve line(0.0, Pnt3Cart(point1.X(), point1.Y(), point1.Z()),
                       1.0, Pnt3Cart(point2.X(), point2.Y(), point2.Z()));

Real integral = PathIntegration::LineIntegral(forceField, line, 0.0, 1.0);

std::cout << "Path integral between point 1 and point 2: " << integral << std::endl;
```

Potential difference based on path integral calculation, along a complex spline curve connecting two points:

```
// second, we'll create a spline parametric curve between those two points
// with some points added in between to make path "curvaceus"
Matrix<Real> curve_points1{ 5, 3,
                            { point1.X(), point1.Y(), point1.Z(),
                             20.0, 5.0, -10.0,
                             100.0, 150.5, -30.0,    // sending it far away
                             5.0, 10.5, 1.0,
                             point2.X(), point2.Y(), point2.Z() }
};
SplineInterpParametricCurve<3> curve(0.0, 1.0, curve_points1);

Real integral2 = PathIntegration::LineIntegral(forceField, curve, 0.0, 1.0);

std::cout << "Path integral along spline curve between point 1 and point 2: " << integral2 << std::endl;
```

Results:

```
Potential at point 1: -6.66667
Potential at point 2: -9.36586
Difference in potential: 2.69919
Path integral between point 1 and point 2: 2.6992
Path integral along spline curve between point 1 and point 2: 2.6992
```

INVESTIGATING LONG-TERM SIMULATION

Show need for symplectic solvers!

GRAVITATIONAL SLINGSHOT – HOW IT WORKS?

9. Simulating gravity in N-body problem

Deep dive with gravity simulation.

Tasks at hand:

- Numerically simulate N-body problem
- Symplectic integrators

SOME THEORY AND COMPUTATIONAL CONSIDERATIONS

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++

We are going to build gravity simulator similarly to previous chapter, but we are going from start for a full 3D case.

Class representing state of N-body system, similar to TwoBodyState, but now all accessors are indexed

```
class NBodyState
{
public:
  std::vector<GravityBodyState> _bodies;

  void addBody(const GravityBodyState& body) { ... }
  void addBody(Real mass, Vec3Cart position) { ... }
  void addBody(Real mass, Vec3Cart position, Vec3Cart velocity) { ... }

  Real     Mass(int i)  const { return _bodies[i].Mass(); }
  Vec3Cart Pos(int i)   const { return _bodies[i].R(); }
  Vec3Cart Vel(int i)   const { return _bodies[i].V(); }

  void SetPosition(int i, Vec3Cart pos) { _bodies[i].R() = pos; }
  void SetVelocity(int i, Vec3Cart vel) { _bodies[i].V() = vel; }
```

And calculators for basic properties

```
Vec3Cart CentreOfMassPos() const { ... }
Vec3Cart CentreOfMassVel() const { ... }

Vec3Cart LinearMomentum() const { ... }
Vec3Cart AngularMomentum(Vec3Cart origin) const { ... }
Vec3Cart AngularMomentumCM() const { ... }
Vec3Cart AngularMomentumOrigin() const { ... }

Real TotalKineticEnergy() const { ... }
Real TotalPotentialEnergy() const { ... }
```

Simulation config

```
class NBodyGravitySimConfig
{
  Real _G;
  std::vector<GravityBodyState> _bodies;

public:
  NBodyGravitySimConfig() : _G(6.67430e-11) {} // Gravitational constant in m^3 kg^-1 s^-2
  NBodyGravitySimConfig(Real G) : _G(G) {}

  Real G() const { return _G; }

  int  NumBodies() const { return (int)_bodies.size(); }

  void AddBody(Real mass, Vec3Cart position, Vec3Cart velocity) { ... }
  void AddBody(Real mass, Vec3Cart position, Vec3Cart velocity, std::string color, Real radius)

  Real  Mass(int i)   const { return _bodies[i].Mass(); }
  Real  Radius(int i) const { return _bodies[i].Radius(); }
  std::string Color(int i)  const { return _bodies[i].Color(); }

  Vec3Cart Position(int i) const { return _bodies[i].R(); }
  Vec3Cart Velocity(int i) const { return _bodies[i].V(); }

  void SetPosition(int i, Vec3Cart pos) { _bodies[i].R() = pos; }
  void SetVelocity(int i, Vec3Cart vel) { _bodies[i].V() = vel; }
};
```

And a class representing ODE for our system

```
class NBodyGravitySystemODE : public IODESystem
{
  NBodyGravitySimConfig _config;

public:
  NBodyGravitySystemODE(NBodyGravitySimConfig inConfig) : _config(inConfig) {}

  int getDim() const { return 6 * _config.NumBodies(); }
  void derivs(const Real t, const Vector<Real>& x, Vector<Real>& dxdt) const { ... }
};
```

Where derivs() function implementation is a little bit more complex (and could be improved in terms of efficiency).

```
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++)
  {
    Vec3Cart force(0, 0, 0);        // calculating force on body i
    for (int j = 0; j < _config.NumBodies(); j++)
    {
      if (i != j)      // checking for self-force
      {
        Vec3Cart vec_dist(x[3 * j]     - x[3 * i],
                          x[3 * j + 1] - x[3 * i + 1],
                          x[3 * j + 2] - x[3 * i + 2]);

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

    // x, y, z coord
    dxdt[3 * i] = x[3 * 2 * i + 1];
    dxdt[3 * i + 1] = x[6 * i + 2];
    dxdt[3 * i + 2] = x[6 * i + 3];

    // vx, vy, vz velocity components
    int half = _config.NumBodies() * 3;
    dxdt[half + 3 * i]     = 1 / _config.Mass(i) * force.X();
    dxdt[half + 3 * i + 1] = 1 / _config.Mass(i) * force.Y();
    dxdt[half + 3 * i + 2] = 1 / _config.Mass(i) * force.Z();
  }
}
```

Class for simulation results

```
class NBodyGravitySimulationResults
{
public:
  const NBodyGravitySimConfig& _config;

  Real _duration;
  std::vector<Real>       _vecTimes;
  std::vector<NBodyState> _vecStates;

  NBodyGravitySimulationResults(const NBodyGravitySimConfig& config) { ... }
  NBodyGravitySimulationResults(const NBodyGravitySimConfig& config, const Real& duration)

  Real        Time(int i)  const { return _vecTimes[i]; }
  NBodyState  State(int i) const { return _vecStates[i]; }

  std::vector<Real>     getTimes() const { return _vecTimes; }
  std::vector<Vec3Cart> getBodyPos(int i) const { ... }
  std::vector<Vec3Cart> getBodyVel(int i) const { ... }
```

We are skipping part, that is also present in this class, with all the utilities as in sim. Results for two body case.

And finally, NBodyGravitySimulator to tie it all together.

```
class NBodyGravitySimulator
{
protected:
  NBodyGravitySimConfig _config;

public:
  NBodyGravitySimulator(NBodyGravitySimConfig inConfig) : _config(inConfig) {}

  NBodyGravitySimulationResults SolveEulerDirect(Real dT, int numSteps) { ... }

  NBodyGravitySimulationResults SolveRK5(Real duration, Real eps, Real minSaveInterval, Real hStart)
};
```

Testing NBodyGravitySimulator

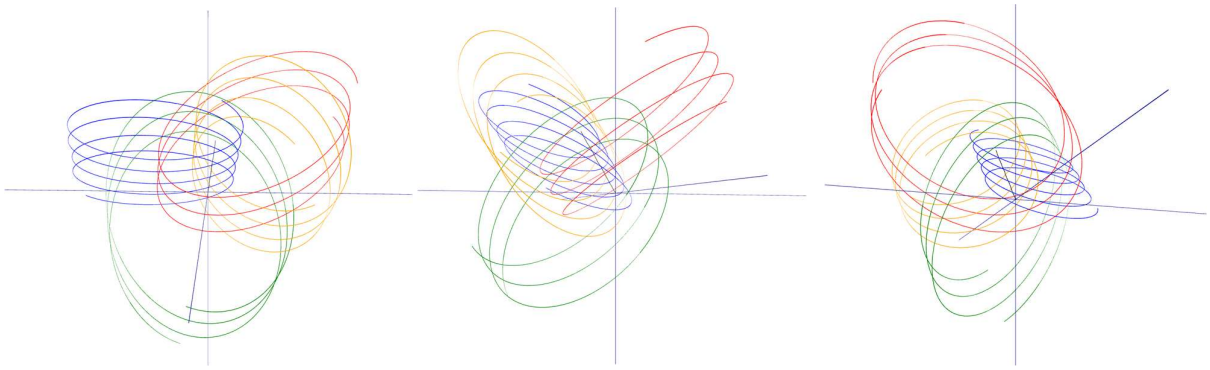
Setting our configurator with five bodies

```
class NBodyGravityConfigGenerator
{
public:
  // configuration with five bodies
  static NBodyGravitySimConfig Config1_five_bodies()
  {
    NBodyGravitySimConfig config(1.0);

    config.AddBody(10000, Vec3Cart{   0.0,    0.0,    0.0 }, Vec3Cart{  0.05, -0.2,  0.3 });
    config.AddBody(   20, Vec3Cart{-110.0,  -50.0,   10.0 }, Vec3Cart{  2.0,   5,   -3.0 });
    config.AddBody(   10, Vec3Cart{ 130.0,   50.0,   70.0 }, Vec3Cart{  2.0,  -5,   -2 });
    config.AddBody(   20, Vec3Cart{ -20.0, -100.0, -110.0 }, Vec3Cart{ -5,    -2.0,  3.0 });
    config.AddBody(   10, Vec3Cart{  70.0, -110.0,   70.0 }, Vec3Cart{ -3,     3,    3.0 });

    return config;
  }
```

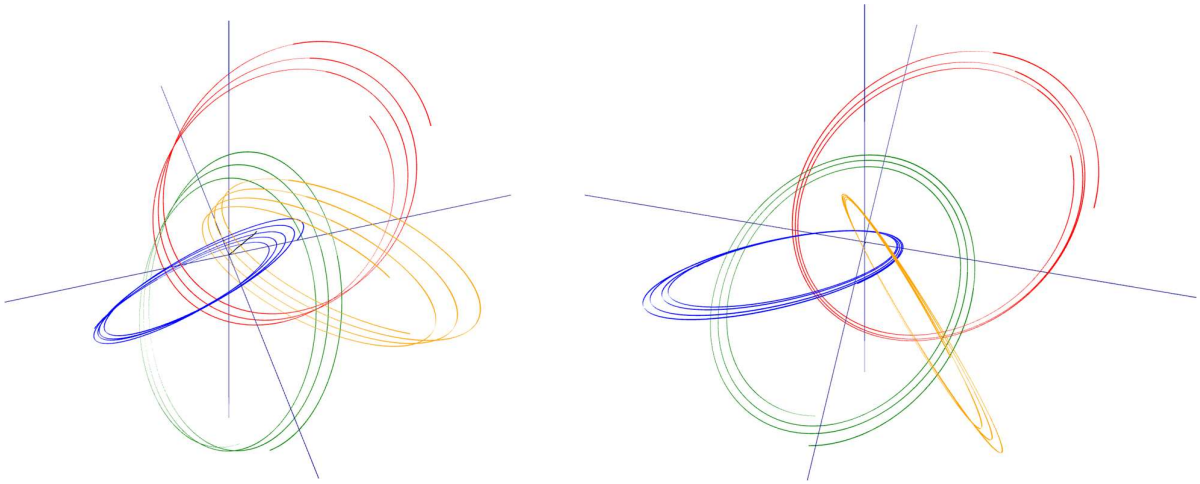
Solving

```
NBodyGravitySimConfig config = NBodyGravityConfigGenerator::Config1_five_bodies();

NBodyGravitySimulator solver(config);

Real t1 = 0.0, t2 = 200.0;

// solving with Euler method
const int  steps = 2001;
const Real dt = (t2 - t1) / steps;
NBodyGravitySimulationResults result = solver.SolveEuler(dt, steps);

// VISUALIZING TRAJECTORIES AS PARAMETRIC CURVES
result.VisualizeAsParamCurve("five_bodies", Vector<int>{ 0, 1, 2, 3, 4 });
```

Results seem chaotic at first.



Until we align our point of view with the movement of central black body, and then we start to see order.



Visualizing fields

SIMULATING GRAVITY IN SOLAR SYSTEM

Configuration is quite a bit more involved

Setting constants and Sun.

```
static NBodyGravitySimConfig Config2_Solar_system()
{
  // Physical constants
  constexpr double G_SI = PhyConst::GravityConstant; // m^3 kg^-1 s^-2
  constexpr double million_km = 1.0e9;     // m
  constexpr double year = 3.15576e7;       // s

  double M_jup = SolarSystem::GetBodyInfo("Jupiter").Mass_kg; // kg
  double M_sun = SolarSystem::GetBodyInfo("Sun").Mass_kg; // kg

  // G in units: [ (million km)^3 / (Jupiter mass * year^2) ]
  double G = G_SI * (M_jup) * (year * year) / (million_km * million_km * million_km);

  NBodyGravitySimConfig config(G);
```

And then planets.

```
// Planets: mass (in Jupiter masses), semi-major axis (in million km)
struct Planet {
  const char* name;
  double mass_Mjup;
  double dist_million_km;
  std::string color;
  double radius;
} planets[] = {
    {"Mercury", 1.6601e-7, 57.91,  "Black", 2.5},
    {"Venus",   2.447e-6,  108.21, "Orange", 6},
    {"Earth",   3.003e-6,  149.60, "Blue", 6},
    {"Mars",    3.227e-7,  227.92, "Red", 4},
    {"Jupiter", 1.0,       778.57, "Brown", 50},
    {"Saturn",  0.299,     1433.53, "Brown", 40},
    {"Uranus",  0.0457,    2872.46, "LightBlue", 20},
    {"Neptune", 0.0539,    4495.06, "Green", 20}
};

double Mtot = M_sun / M_jup;
for (const auto& p : planets) Mtot += p.mass_Mjup;
```

And filling up velocities.

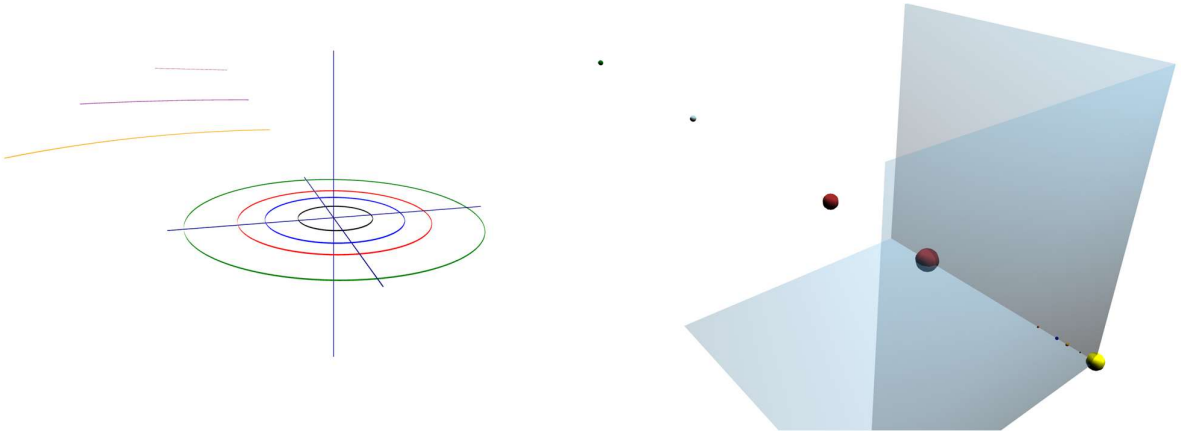
```
  for (const auto& p : planets)
  {
    // Circular orbit velocity (in million km/year)
    // v = sqrt(G * M_sun/M_jup / r)
    double v = std::sqrt(G * (M_sun / M_jup) / p.dist_million_km);

    // Place planet at (a, 0, 0), velocity (0, v, 0)
    config.AddBody(
      p.mass_Mjup,
      Vec3Cart{ p.dist_million_km, 0, 0 },
      Vec3Cart{ 0, v, 10 },
      p.color,
      p.radius
    );
  }

  return config;
}
```

Simulation function is simple.

```
void Demo_Solar_system()
{
  NBodyGravitySimConfig config = NBodyGravityConfigGenerator::Config2_Solar_system();

  NBodyGravitySimulator solver(config);

  Real t1 = 0.0, t2 = 2;
  const int  steps = 501;
  const Real dt = (t2 - t1) / steps;

  // solving with Euler method
  NBodyGravitySimulationResults result = solver.SolveEuler(dt, steps);

  result.VisualizeAsParamCurve("solar_system", Vector<int>{ 1, 2, 3, 4, 5 });

  // visualize as particle simulation
  result.VisualizeAsParticleSimulation("solar_system_particles", Vector<int>{ 1, 2, 3, 4, 5 }, dt);
}
```

And we get.

Visualized as orbital parametric curves and as particle simulations.



Where radiuses of planets are show to relative scale (ie. Jupiter is indeed that much bigger than blue Earth), but are also increased couple million/billion times relative to the scale of distance between planets.

Also, in this scale, Sun would take half the picture.

Simulating Voyager path through Solar system

SIMULATING COLLISION OF STAR CLUSTERS

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

INTRODUCING COORDINATE SYSTEMS

Primjer Manhattan na GPS coordinate

Nije lako, ali može se!

No, ključno, s koordinatama ulica, pa čak I mapiranjem svakog stana I njegove adrese na GPS koordinatu, ne može se ništa matematički

Da bi se moglo, treba mapirati na neki lokalni Cartesian system … ili skup takvih Sistema, međusobno konitnuirano povezanih

Primjer polar2d, spherical, cylindrical

TRANSFORMATION OF COORDINATES

General transformation of coordinates:

$$\begin{cases} \bar{x}^1 = f^1(x^1, \ldots, x^n), \\ \ldots\ldots\ldots\ldots\ldots\ldots, \\ \bar{x}^n = f^n(x^1, \ldots, x^n), \end{cases}$$

Or in short notation,

$$\bar{x}^j = f^j(x^h) \qquad (j = 1, \ldots, n).$$

It is assumed we also have an inverse transformation.

Different characterizations/types of coordinate systems

- Homogenous / non-homogenous
- Orthogonal / oblique
- Linear / curvilinear

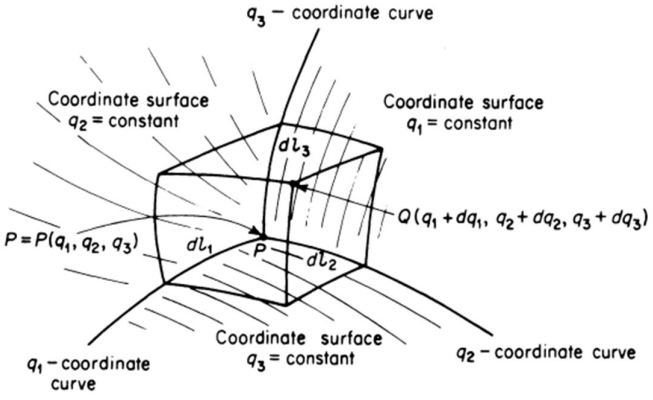
Polar coordinates in 2D

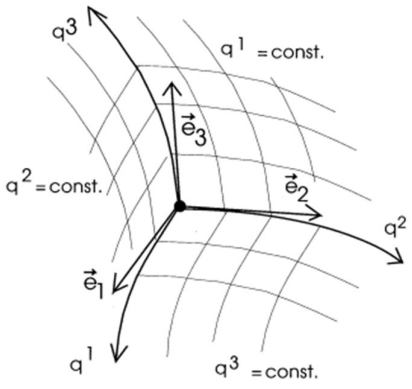
Rotations in 2D

Orthogonal transformations

Curvilinear

Visualization of general curvilinear coordinate system





Spherical

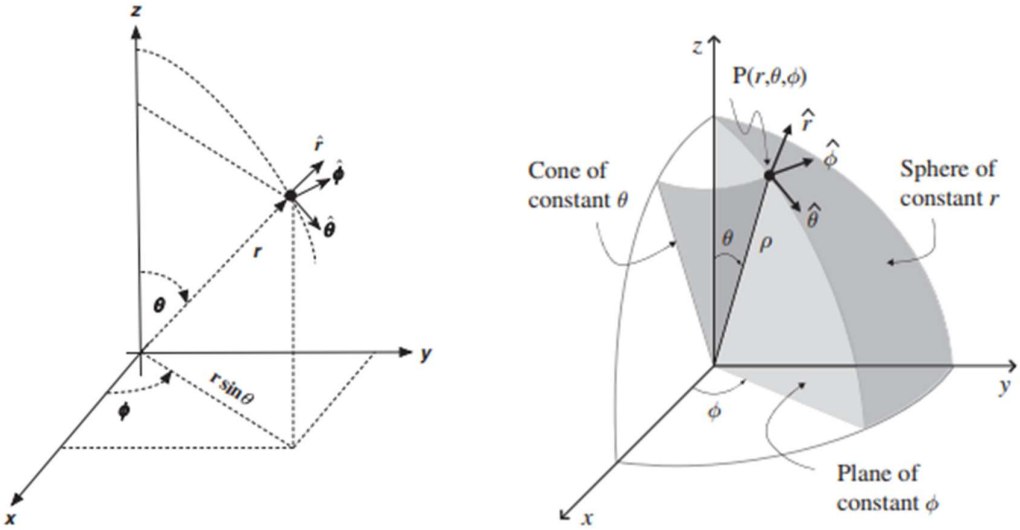


Figure 23. Spherical system coordinates

Transformation formulas. First, from a set of Cartesian coordinates (x, y, z) to spherical (r, θ, ϕ)

$$r = \sqrt{x^2 + y^2 + z^2}$$
$$\theta = \cos^{-1}(z/r)$$
$$\phi = \tan^{-1}(y/x),$$

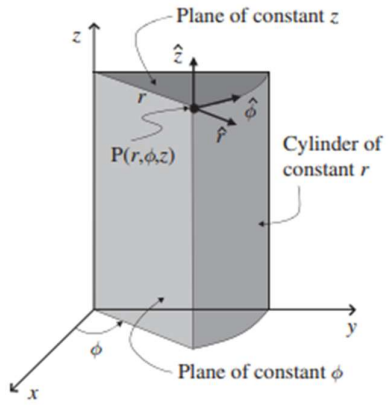
And the inverse transformation, from spherical to Cartesian.

$$x = r \sin\theta \cos\phi$$
$$y = r \sin\theta \sin\phi$$
$$z = r \cos\theta.$$

TODO - Mathematical and physical convention in ordering coordinates.

Cylindrical

Cylindrical coordinate system is extension of 2D polar coordinate system to 3 dimensions, by simply including z as third coordinate.



Transformation from Cartesian to cylindrical.

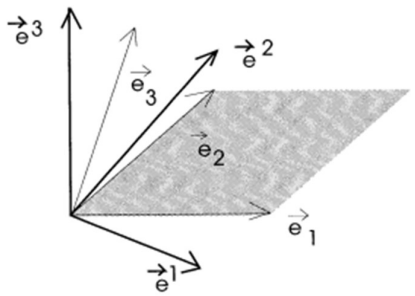
$$r = \sqrt{x^2 + y^2}$$
$$\theta = \arctan\left(\frac{y}{x}\right)$$
$$z = z$$

And inverse.

$$x = r\cos(\phi)$$
$$y = r\sin(\phi)$$
$$z = z.$$

Oblique Cartesian coordinate system

Dual basis



Constructing reciprocal (dual) basis in 3D.

TRANSFORMING VECTORS

Covariant and contravariant vectors

TRANSFORMING TENSORS

IMPLEMENTING COORDINATE TRANSFORMATIONS 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, that will be base classes for all our coordinate transformations.

```
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) { ... }
};
```

These are still abstract classes, expecting from inheriting classes to provide implementations from interface, but based on those implementations, they provide following functionality.

First, some helper classes.

Vector3Spherical

```
class Vector3Spherical : public VectorN<Real, 3>
{
public:
  Real  R()     const { return _val[0]; }
  Real& R()           { return _val[0]; }
  Real  Theta() const { return _val[1]; }
  Real& Theta()       { return _val[1]; }
  Real  Phi()   const { return _val[2]; }
  Real& Phi()         { return _val[2]; }

  Vector3Spherical() : VectorN<Real, 3>{ 0.0, 0.0, 0.0 } {}
  Vector3Spherical(const VectorN<Real, 3>& b) : VectorN<Real, 3>{ b[0], b[1], b[2] } {}
  Vector3Spherical(Real r, Real theta, Real phi) : VectorN<Real, 3>{ r, theta, phi } {}
  Vector3Spherical(std::initializer_list<Real> list) : VectorN<Real, 3>(list) { }

  Vector3Spherical GetAsUnitVectorAtPos(const Vector3Spherical& pos) const { ... }

  std::ostream& PrintDeg(std::ostream& stream, int width, int precision) const { ... }
};
```

Vector3Cylindrical

```
class Vector3Cylindrical : public VectorN<Real, 3>
{
public:
  Real    R()   const { return _val[0]; }
  Real& R()           { return _val[0]; }
  Real    Phi() const { return _val[1]; }
  Real& Phi()         { return _val[1]; }
  Real    Z()   const { return _val[2]; }
  Real& Z()           { return _val[2]; }

  Vector3Cylindrical() : VectorN<Real, 3>{ 0.0, 0.0, 0.0 } {}
  Vector3Cylindrical(const VectorN<Real, 3>& b) : VectorN<Real, 3>{ b[0], b[1], b[2] } {}
  Vector3Cylindrical(Real r, Real phi, Real z) : VectorN<Real, 3>{ r, phi, z } {}
  Vector3Cylindrical(std::initializer_list<Real> list) : VectorN<Real, 3>(list) { }

  Vector3Cylindrical GetAsUnitVectorAtPos(const Vector3Cylindrical& pos) const { ... }
};
```

Transforming vectors

Essence of this classes, and main reason for their existence, is in the set of 'transf' functions, where we implement general transformation rules for vectors (and tensors), that will be available in all classes that inherit from this abstract classes (and implement necessary interfaces).

```
VectorTo   transfVecContravariant(const VectorFrom& vec, const VectorFrom& pos)
{
  VectorFrom ret;
  for (int i = 0; i < N; i++) {
    ret[i] = 0;
    for (int j = 0; j < N; j++)
      ret[i] += Derivation::NDer4Partial(this->coordTransfFunc(i), j, pos) * vec[j];
  }
  return ret;
}
```

And also, transformation of covariant vector:

```
VectorTo   transfVecCovariant(const VectorFrom& vec, const VectorTo& pos)
{
  VectorTo ret;
  for (int i = 0; i < N; i++)
  {
    ret[i] = 0;
    for (int j = 0; j < N; j++)
      ret[i] += Derivation::NDer4Partial(this->inverseCoordTransfFunc(j), i, pos) * vec[j];
  }

  return ret;
}
```

Transforming tensors

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;
}
```

EXAMPLE IMPLEMENTATIONS OF COORDINATE TRANSFORMATIONS

First, the static ones

Polar coordinates transformation

Spherical coordinates transformation

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]; }
```

Rotations in 2D

Rotations in 3D

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;
```

For coordinate transformations that depend on parameters (and so can't be made completely static), 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);
}
```

TODO – finish example

Oblique 2D Cartesian transformation

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 centripetal forces on a simple 2D carousel

PHYSICS OF ROTATING SYSTEMS

SIMPLE CAROUSEL AND CENTRIFUGAL FORCE

Carousel with radius of 20 meters, throwing darts

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

INTRODUCING REFERENTIALFRAME

Carousel on carousel ☺

Shooting method za ODE's?

12. Projectile/rocket 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 35 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

NUMERICALLY CALCULATING MOMENT OF INERTIA TENSOR

Calculating moment of inertia tensor 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 tensor for continuous mass

Models:

```
// continuous mass with defined boundaries (needs density function)
class IContinuousMass
{
public:
    Real _x1, _x2;

    Real (*_y1)(Real);
    Real (*_y2)(Real);

    Real (*_z1)(Real, Real);
    Real (*_z2)(Real, Real);

    IContinuousMass(Real x1, Real x2, Real (*y1)(Real), Real (*y2)(Real),
                    Real (*z1)(Real, Real), Real (*z2)(Real, Real)) { ... }

    virtual Real getDensity(const VectorN<Real, 3> &x) const = 0;
};
```

```
class ContinuousMass : public IContinuousMass
{
    Real (*_density)(const VectorN<Real, 3> &x);
public:
    ContinuousMass(Real x1, Real x2, Real (*y1)(Real), Real (*y2)(Real),
                   Real (*z1)(Real, Real), Real (*z2)(Real, Real),
                   Real (*density)(const VectorN<Real, 3> &x)) { ... }

    virtual Real getDensity(const VectorN<Real, 3> &x) const { ... }
};
```

```
class ContinuousMassConstDensity : public IContinuousMass
{
    Real _density;
public:
    ContinuousMassConstDensity(Real x1, Real x2, Real (*y1)(Real), Real (*y2)(Real),
                               Real (*z1)(Real, Real), Real (*z2)(Real, Real),
                               Real density) { ... }

    virtual Real getDensity(const VectorN<Real, 3> &x) const { ... }
};
```

Base class for defining functions that we need to integrate for specific tensor components

```
class ContMassScalarFuncBase : public IScalarFunction<3>
{
protected:
  IContinuousMass &_mass;
public:
  ContMassScalarFuncBase(IContinuousMass& mass) : _mass(mass) {}
};
```

6 implementations for 6 independent components of moment of inertia.

```
struct Func11 : public ContMassScalarFuncBase
{
  Func11(IContinuousMass& mass) : ContMassScalarFuncBase(mass) {}
  Real operator()(const VectorN<Real, 3>& x) const
  {
    return _mass.getDensity(x) * (x[1]*x[1] + x[2]*x[2]);
  }
};
struct Func22 { ... };
struct Func33 { ... };
struct Func12 { ... };
struct Func13 : public ContMassScalarFuncBase
{
  Func13(IContinuousMass& mass) : ContMassScalarFuncBase(mass) {}
  Real operator()(const VectorN<Real, 3>& x) const
  {
    return -_mass.getDensity(x) * x[0] * x[2];
  }
};
struct Func23 { ... };
```

Calculator class

```
class ContinuousMassMomentOfInertiaTensorCalculator
{
    IContinuousMass &_mass;
public:
    ContinuousMassMomentOfInertiaTensorCalculator(IContinuousMass &mass) { ... }

    Tensor2<3> calculate() { ... }
};
```

Implementation of calculation

```
Tensor2<3> calculate()
{
    Tensor2<3> tensor(2,0);

    Func11 _f11(_mass);
    Func22 _f22(_mass);
    Func33 _f33(_mass);
    Func12 _f12(_mass);
    Func13 _f13(_mass);
    Func23 _f23(_mass);
    tensor(0, 0) = Integrate3D(_f11, _mass._x1, _mass._x2, _mass._y1, _mass._y2, _mass._z1, _mass._z2);
    tensor(1, 1) = Integrate3D(_f22, _mass._x1, _mass._x2, _mass._y1, _mass._y2, _mass._z1, _mass._z2);
    tensor(2, 2) = Integrate3D(_f33, _mass._x1, _mass._x2, _mass._y1, _mass._y2, _mass._z1, _mass._z2);
    tensor(0, 1) = Integrate3D(_f12, _mass._x1, _mass._x2, _mass._y1, _mass._y2, _mass._z1, _mass._z2);
    tensor(0, 2) = Integrate3D(_f13, _mass._x1, _mass._x2, _mass._y1, _mass._y2, _mass._z1, _mass._z2);
    tensor(1, 2) = Integrate3D(_f23, _mass._x1, _mass._x2, _mass._y1, _mass._y2, _mass._z1, _mass._z2);
    tensor(1, 0) = tensor(0, 1);
    tensor(2, 0) = tensor(0, 2);
    tensor(2, 1) = tensor(1, 2);

    return tensor;
}
```

As an example, this is how to calculate moment of inertia tensor for cube.

```
// create ContinuousMass representation for cube of constant density
SolidBodyWithBoundaryConstDensity cube(-0.5, 0.5,
  [](Real x) { return -0.5; },
  [](Real x) { return 0.5; },
  [](Real x, Real y) { return -0.5; },
  [](Real x, Real y) { return 0.5; },
  1.0);

ContinuousMassMomentOfInertiaTensorCalculator calculator(cube);

Tensor2<3> tensor = calculator.calculate();

std::cout << "Tensor of inertia for cube: " << std::endl;
```

```
Tensor of inertia for cube:
(N = 3)
[    0.1666666667,    0.0000000000,    0.0000000000,  ]
[    0.0000000000,    0.1666666667,    0.0000000000,  ]
[    0.0000000000,    0.0000000000,    0.1666666667,  ]
```

And sphere.

```
SolidBodyWithBoundaryConstDensity sphere(-1.0, 1.0,
  [](Real x) { return -sqrt(1 - x * x); },
  [](Real x) { return sqrt(1 - x * x); },
  [](Real x, Real y) { return -sqrt(1 - x * x - y * y); },
  [](Real x, Real y) { return sqrt(1 - x * x - y * y); },
  1.0);

ContinuousMassMomentOfInertiaTensorCalculator calculator2(sphere);

Tensor2<3> tensor2 = calculator2.calculate();

std::cout << "Tensor of inertia for sphere: " << std::endl;
std::cout << tensor2 << std::endl;

std::cout << "Theoretical value for sphere: " << std::endl;
std::cout << 2.0 / 5.0 * (4.0 / 3.0 * Constants::PI) << std::endl;
```

```
Tensor of inertia for sphere:
(N = 3)
[    1.6770777757,    0.0000000000,    0.0000000000,  ]
[    0.0000000000,    1.6758834411,    0.0000000000,  ]
[    0.0000000000,    0.0000000000,    1.6774758872,  ]

Theoretical value for sphere:
1.6755160819
```

Testing tensor transformation laws

Moment of inertia tensor will be useful to as a first example where we can verify implemented functionality for tensor transformations.

Setup of discrete masses

```
// define set of discrete masses
double a = 1;
Vector3Cartesian pos1(a, a, 0);
Vector3Cartesian pos2(-a, a, 0);
Vector3Cartesian pos3(-a, -a, 0);
Vector3Cartesian pos4(a, -a, 0);
Vector3Cartesian pos5(0, 0, 4 * a);
double m1 = 1;
double m2 = 1;
double m3 = 1;
double m4 = 1;
double m5 = 1;

DiscreteMass mass1(pos1, m1);
DiscreteMass mass2(pos2, m2);
DiscreteMass mass3(pos3, m3);
DiscreteMass mass4(pos4, m4);
DiscreteMass mass5(pos5, m5);

std::vector<DiscreteMass> masses = { mass1, mass2, mass3, mass4, mass5 };
DiscreteMassesConfig massesConfig(masses);
```

Calculating moment of inertia for original configuration

```
DiscreteMassMomentOfInertiaTensorCalculator calculator(massesConfig);
Tensor2<3> tensor_orig = calculator.calculate();

std::cout << "Tensor of inertia: " << std::endl;
std::cout << tensor_orig << std::endl;
```

```
Tensor of inertia:
(N = 3)
[   20.0000000000,    0.0000000000,    0.0000000000,  ]
[    0.0000000000,   20.0000000000,    0.0000000000,  ]
[    0.0000000000,    0.0000000000,    8.0000000000,  ]
```

Now we will investigate what happens if we change coord.system, in two cases:

1. using coord.system transform we calculate TRANSFORMED (original) tensor

2. using coord.system transform we calculate NEW set of masses and then calculate tensor

```
// new coord.system is rotated around x axis for 30 degrees
CoordTransfCart3DRotationXAxis coord_transf(Utils::DegToRad(30.0));
```

```
// 1) - calculated using tensor transformation
Tensor2<3> tensor_transf = coord_transf.transfTensor2(tensor_orig, Vector3Cartesian(1, 1, 1));

std::cout << "Tensor of inertia transformed: " << std::endl;
std::cout << tensor_transf << std::endl;
```

```
Tensor of inertia transformed:
(N = 3)
[   20.0000000000,    0.0000000000,    0.0000000000,  ]
[    0.0000000000,   16.9999999628,    5.1961524318,  ]
[    0.0000000000,    5.1961524318,   10.9999999911,  ]
```

```cpp
// 2) - change masses position and calculate new tensor directly
DiscreteMassesConfig massesTransConfig(masses);
for (auto& mass : massesTransConfig._masses)
  mass._position = coord_transf.transf(mass._position);

DiscreteMassMomentOfInertiaTensorCalculator calculator2(massesTransConfig);
Tensor2<3> tensor_changed = calculator2.calculate();

std::cout << "Tensor of inertia rotated masses: " << std::endl;
std::cout << tensor_changed << std::endl;
```

```
Tensor of inertia rotated masses:
(N = 3)
[   20.0000000000,    0.0000000000,    0.0000000000,  ]
[    0.0000000000,   17.0000000000,    5.1961524227,  ]
[    0.0000000000,    5.1961524227,   11.0000000000,  ]
```

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 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

```
class CoordTransfLorentzXAxis : public CoordTransfWithInverse<Vector4Lorentz, Vector4Lorentz, 4>
{
private:
  Real    _velocity;      // expressed in units of c (speed of light)
  MatrixNM<Real, 4, 4>  _transf;
  MatrixNM<Real, 4, 4>  _inverse;

  const ScalarFunctionFromStdFunc<4> _f1, _f2, _f3, _f4;
  const ScalarFunctionFromStdFunc<4> _fInv1, _fInv2, _fInv3, _fInv4;
```

Monstruous constructor, setting everything up for of transformation function

```
CoordTransfLorentzXAxis(Real inVelocity) : _velocity(inVelocity),
  _f1(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::func1, this, std::placeholders::_1) }),
  _f2(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::func2, this, std::placeholders::_1) }),
  _f3(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::func3, this, std::placeholders::_1) }),
  _f4(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::func4, this, std::placeholders::_1) }),
  _fInv1(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::funcInverse1, this, std::placeholder
  _fInv2(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::funcInverse2, this, std::placeholder
  _fInv3(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::funcInverse3, this, std::placeholder
  _fInv4(std::function<Real(const VectorN<Real, 4>&)> { std::bind(&CoordTransfLorentzXAxis::funcInverse4, this, std::placeholder
{
  double gamma = 1.0 / sqrt(1.0 - _velocity * _velocity);
  double beta = _velocity;

  _transf[0][0] = gamma;
  _transf[0][1] = -beta * gamma;
  _transf[1][0] = -beta * gamma;
  _transf[1][1] = gamma;
  _transf[2][2] = 1.0;
  _transf[3][3] = 1.0;

  _inverse[0][0] = gamma;
  _inverse[0][1] = -beta * gamma;
  _inverse[1][0] = -beta * gamma;
  _inverse[1][1] = gamma;
  _inverse[2][2] = 1.0;
  _inverse[3][3] = 1.0;
}
```

MATH - INTRODUCING METRIC TENSOR

Implementing metric tensors in C++

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 thing is where that projectile will appear (!) to be, to Earth observer at the location where it was fired from.

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 (short) investigations in electromagnetism.

Task at hand:

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

PHYSICS – COULOMB LAW

Jednostavno numeričko izračunavanje potencijala za analitički poznata rješenja I usporedba

Vizualizacija skupa single charge potencijala I sila

SIMULATING DISTRIBUTION OF CHARGE ON A SOLID BODY

What is distribution of charge on different solid bodies?

Sfera

Cilindar

Kvadar

Something with “pointy” end

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

Modeling curves 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

Mathematics of surfaces

Modeling surfaces in C++

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

SIMULATING TRAVEL BEYOND EVENT HORIZON

REFERENCES

01 - Physics

Mechanics

- Mechanics, 3rd Ed, Landau, Lifshitz
- Classical Dynamics of Particles and Systems, S. Thornton, J. Marion, Thomson (2004)
- Theoretical mechanics of particles and continua, Alexander L.Fetter, John D.Walecka, Dover, (2003)
- Analytical Mechanics, Sergio Cecotti, Springer Nature Switzerland (2024)

Gravity

Electromagnetism

Special relativity

General relativity

02 - Mathematical methods in physics

- Mathematical Methods for Physicists, 7th Ed, A Comprehensive Guide, Arfken, George Brown Harris, Frank E. Weber, Hans-Jurgen, Elsevier_Academic Press (2013)
- A guided tour of mathematical methods for the physical sciences, 3rd Ed, Snieder, Roel Van Wijk, Kasper, Cambridge University Press (2015)
- Mathematical Physics, Eugene Butkov, Addison-Wesley (1968)
- Mathematical methods for physics and engineering, 3rd Edition, K. F. Riley, M. P. Hobson, S. J. Bence, Cambridge University Press (2006)

03 - Numerical analysis and mathematics

- A Survey of Numerical Mathematics, Vol 1, David M Young, Robert Todd Gregory, Addison-Wesley Pub. Co (1972-73)
- A survey of numerical mathematics, Vol 2, David M. Young, Robert T. Gregory, Addison-Wesley Educational Publishers Inc (1973)

04 - General numerical & scientific computing

- Numerical Recipes - The Art of Scientific Computing, 3rd Ed, William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Cambridge University Press (2007)
- Introduction to numerical programming - a practical guide for scientists and engineers using Python and C/C++, Titus A. Beu, CRC Press (2015)

05 - C++

06 - Computational physics – general

- Computational Physics - An Introduction, Franz J. Vesely (auth.), Springer US (2001)
- Computational Physics, Jos Thijssen, Cambridge University Press (2012)
- Computational Physics, Nicholas J. Giordano, Hisao Nakanishi, Addison-Wesley (2005)
- Computational physics _ simulation of classical and quantum systems, Scherer, Philipp O. J, Springer (2017)

07 - Computational physics - C++

- Game physics, Eberly, David H, CRC Press (2015)

08 - Computational physics - Python

- Computational Physics - Problem Solving with Python, 4th Edition, Rubin H. Landau, Manuel J. Páez etc., (2024)

- Numerical Methods in Physics with Python, 2nd Edition, Alex Gezerlis, Cambridge University Press (2023)
- Programming For Computations - Python A Gentle Introduction To Numerical Simulations With Python 3.6, 2nd Ed, Svein Linge, Hans Petter Langtangen (2020)
- Applied Numerical Methods with Python for Engineers and Scientists, Steven Chapra, David Clough, McGraw Hill (2021)
-

09 - Computational physics - Matlab & Mathematica