VECTORS MATRICES AND C++ CODE

- Professionally typeset
- Basic level.
- Hundreds of live crossreferences
- Live Index
- Live Table of Contents
- Live Bibliography
- Live Internet links
- Live C++ code documentation
- Professional C++ code included
- New to C++? C++ Course included

Sergio Pissanetzky

Vectors, Matrices, and C++ Code

Sergio Pissanetzky

2004

Professionally typeset by LaTeX

Dr. Pissanetzky retired after a rewarding career as an Entrepreneur, Professor, Research Scientist and Consultant. He was the founder of Magnus Software Corporation, where he focused on development of specialized applications for the Magnetic Resonance Imaging (MRI) and the High Energy Particle Accelerator industries. He has served as Member of the International Editorial Board of the "International Journal for Computation in Electrical and Electronic Engineering", as a Member of the International Advisory Committee of the International Journal "Métodos Numéricos para Cálculo y Diseño en Ingeniería", and as a member of the International Committee for Nuclear Resonance Spectroscopy, Tokyo, Japan. Dr. Pissanetzky has held professorships in Physics at Texas A&M University and the Universities of Buenos Aires, Córdoba and Cuyo, Argentina. He has also held positions as a Research Scientist with the Houston Advanced Research Center, as Chairman of the Computer Center of the Atomic Energy Commission, San Carlos de Bariloche, Argentina, and as a Scientific Consultant at Brookhaven National Laboratory. Dr. Pissanetzky is currently a member of the Advisory Board of Meedio, LLC. Dr. Pissanetzky holds several US and European patents and is the author of two books and numerous peer reviewed technical papers. Dr.Pissanetzky earned his Ph.D. in Physics at the Balseiro Institute, Univesity of Cuyo, in 1965. Dr. Pissanetzky has 35 years of teaching experience and 30 years of programming experience in languages such as Fortran, Basic, C and C++. Dr. Pissanetzky now lives in a quite suburban neighborhood in Texas.

Website: `http://www.SciControls.com`

ISBN 0-00-000000-0

Preface

Presented here is an integrated approach - perhaps the first in its class - of the basics of vector and matrix Algebra with the object-oriented C++ code that implements the vectors and matrix objects and brings them to life. This is not the traditional road map followed by textbooks, and a rationale is needed.

The concept of object has existed in Science for centuries. An object associates properties and behavior in a single, inseparable entity, an abstraction consisting of properties and a description of their behavior. Although the term *object* is not used in Science, scientists always did all their thinking in terms of objects. In Physics, a rigid body, an atom, an electromagnetic wave, are all objects, because they all have properties and behavior. The laws of Physics describe their behavior in terms of their properties. Similarly, in Mathematics, a vector, a graph, even a number, all have properties and behavior and are therefore objects.

As a matter of fact, thinking in terms of objects is the natural way of thinking, perhaps the only way of thinking, and we all practice it every day. A piece of paper is an object, it has color, size, weight, shape. It can be printed upon, it can absorb water, it can burn, it can be folded or torn and hold its new shape. A bank account is an object, it has a balance and rules that govern how funds can be deposited, withdrawn or transferred, and how interest is earned. Man knew objects for as long as humanity existed, primitive objects such as food or shelter, and more technological objects such as fire or the wheel.

More recently, the concept of object was introduced in Computation, and object-oriented languages were created. A computational object is a model of the real object, an abstraction. So also are our thoughts. In either case, creating the object is difficult because the concept of modeling or abstracting the real world is involved. Among the new languages, C++ became very popular, and is our choice because it offers full and efficient support for all the mathematical calculations we need, and it is entirely built around object-oriented concepts.

Yet, the concept of object is not routinely used when teaching Science, frequently not even mentioned. And the idea that objects can come alive in a computer remains relegated to specialized areas of Computer Science and has not been fully exploited.

As a result, the world of disciplines is divided into the world of science and the world of code. Books on science are written by scientists who may not be very used to coding, software documentation and books on coding are written by developers and computer scientists who may not be very interested in Mathematics or Physics. And books on Computational Physics tend to concentrate on the specialized mathematical methods used to solve the problems but offer at best a background of the mathematical and physical concepts and little or nothing about the code. Many simply refer the reader to a commercial implementation of the methods, thus neatly separating the two worlds.

Where is the scientist who has never felt the need to calculate some numbers in support of his/her research? Where is the developer who has never felt the need for science when writing code? Where is the user who has never been confused with code documentation written by someone not very familiar with a right-handed coordinate system or a 0-based array index?

This book is intended to provide an integrated approach to basic vector and matrix Algebra with object-oriented concepts and the actual code implementing them. The source code is included and readers are free to use it for their own work provided proper credit is always given. Coding is about empowerment, and we want to empower the reader with the ability to create his/her own live objects and cause them to act their parts. This is the first volume of what we expect to be a series covering, approximately, the following topics of Mathematics and Physics: Coordinate Transformations, Graphs, Sparse Matrices [3], Linear Equations, the Dynamics of Multibody Systems, and the intelligent control of Articulated Multibody Robotic Structures. Other titles may be added as needed. All of them, of course, with the corresponding source code included.

This product is both a textbook and a software release. The book can be regarded as very complete software documentation, consisting not only of the description of classes, attributes and methods, but also of the mathematical background that supports the code. Just as business code is best understood by those with a background in that business, scientific code is best understood by those with a background in mathematics.

The code presented here is in no way new. Its roots date back more than 30 years. It evolved from its original non-object-oriented Fortran versions to its current fully object-oriented versions in C++. It grew as it evolved, becoming part of several well known professional scientific programs such as Kubik [5], Magnus [6], Epilog [7], PhysicSolver [9], and others, and it served engineering applications such as the design of Particle Accelerators, Magnetic Resonance Imaging systems, thermal systems for nuclear reactors, and many others, and the teaching of Science.

To understand this book you will need a basic knowledge of Mathematical notation, Algebra and Trigonometry. If you are not familiar with C++ or object-oriented methodology, you can learn it right here, because a basic course on C++ and computational objects is included. It is not a regular course, because as you learn you can refer directly to the included professional code, something not usually available in a traditional course. If you are not interested in learning vectors and matrices but only in using the code, you can still do so. The code documentation has links to the underlying mathematical concepts, or it can be understood even if you ignore them. We do encourage you to check the concepts, however, you can learn some Mathematics in the process. You should read this book if:

- You are a developer and you need a background in vector or matrix algebra.
- You are a science student and you need to learn C++.
- You are a scientist or a science student and you need to write advanced code but you don't want to waste time developing the basics.
- You need ready-to-use C++ source code for your science project.

We could have released our material as html web pages to the Internet, but we have chosen to present it in pdf format and in the form of an electronic book, an eBook. Web pages are sometimes disconcerting, and the navigational adventure may be chaotic because the emphasis in light, color and motion creates confusion. One can write a whole encyclopedia in web pages and nobody will ever realize the magnitude or comprehend the extent or content of the work. Concepts are scattered and difficult to relate to each other, or even to find.

EBooks, instead, offer unity and integrity of content. The reader travels between boundaries with a clear notion of content, quality and organization of the information. Modern eBooks have all the live cross references and internal and external links that one finds in web pages. EBooks are free from advertising, and they can talk. EBooks are sturdy, pages never wear out or get loose. Our eBook has a live table of contents, a live index, live bibliography, live references, live comments, and external links to the Internet.

There are currently thousands of eBooks covering many areas of human knowledge or entertainment. In a sense, eBooks are changing the very art of technical book writing. Using links and cross references, it has now become possible to include advanced and even encyclopedic content without disturbing the natural flow of the subject, something not found in paper textbooks, and something authors are not yet accustomed to. EBooks are even ideal to be used as a reference for the subject matter because of the many live internal links.

If you are reading this, you are probably using the Adobe Reader. Just a couple of suggestions you may find useful. Run the reader by itself, not inside a browser, to maximize the amount of screen area available to you. Magnify as much as you can to improve readability. But try to see a whole page on the screen, even if a piece of the Reader is outside the screen. You can use the left and right arrows in your keyboard to change pages. Internal links will also bring you to another page, and you can return to where you were using alt-left arrow, even repetitively. Ctlr-2 sets fit-to-page and lets you adjust the magnification by dragging the edge of the window until the height of the page is as tall as fits in the window. Also ctrl-m is very useful, it lets you type-in the magnification you want. There are other good pdf readers as well, for example GSview from Ghostscirpt.

Please drop us a line when you feel like it, just make sure your e-mail is clearly identified so it doesn't get confused with spam. We will try to respond via frequently asked questions in our web site, so please read that material before you write. Please consider that we are a few, you are many. Enjoy!

Sergio Pissanetzky
October 2004

Contents

Now let **a** and **b** be two given vectors, which we can express in components as was done in equation 2.7. The dot product is given by:

$$\mathbf{a} \cdot \mathbf{b} = (a_x\,\mathbf{i} + a_y\,\mathbf{j} + a_z\,\mathbf{k}) \cdot (b_x\,\mathbf{i} + b_y\,\mathbf{j} + b_z\,\mathbf{k}) \tag{2.20}$$

Using the associative property to expand this, and using equations 2.19 to calculate the values of several resulting dot products, we obtain:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \tag{2.21}$$

Equation 2.21 tells us how to calculate the dot product between two vectors directly from their components. The dot product of a vector by itself is the square of its magnitude.

2.8 Cross product

The *cross product*, also known as the *vector product*, of two given vectors **a** and **b**, is represented as $\mathbf{a} \times \mathbf{b}$, and is defined as another vector **c**:

$$\mathbf{c} = \mathbf{a} \times \mathbf{b} \tag{2.22}$$

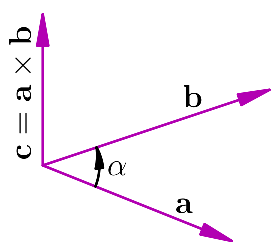

Figure 2.10: The cross product.

Vector **c** is perpendicular to the plane determined by vectors **a** and **b**, its direction is determined in such a way that the three vectors **a**, **b** and **c** form a right-handed system, and its magnitude is the product of the magnitude of **a**, the magnitude of **b**, and the sine of the angle between the positive directions of **a** and **b**:

$$c = a\,b\,\sin(\alpha) \tag{2.23}$$

where α is the angle between **a** and **b**. Figure 2.10 illustrates an example. If **a** and **b** are parallel or antiparallel, the result of the vector product is the zero vector. This is because $\alpha = 0$ or $\alpha = \pi$, respectively, and $\sin\alpha = 0$ in both cases. If **a** and **b** are not parallel, then their directions do in and the elements of $\mathbf{C}$ are those of $\mathbf{A}$ multiplied by s:

$$C_{ij} = sA_{ij} \tag{3.8}$$

for all i and j. The matrix and the number commute:

$$s\mathbf{A} = \mathbf{A}s \tag{3.9}$$

The operation of multiplication can be defined for two matrices of any size or shape, provided they *conform* to each other in the order they are given. Two matrices $\mathbf{A}$ and $\mathbf{B}$ are said to conform in that order if the number of columns of $\mathbf{A}$ is equal to the number of rows of $\mathbf{B}$. Let $\mathbf{A}$ be a $p \times q$ matrix, and let $\mathbf{B}$ be a $q \times r$ matrix. Then matrix $\mathbf{C}$ is the product of $\mathbf{A}$ and $\mathbf{B}$

$$\underset{p \times r}{\mathbf{C}} = \underset{p \times q}{\mathbf{A}} \; \underset{q \times r}{\mathbf{B}} \tag{3.10}$$

if $\mathbf{C}$ is of order $p \times r$ and its elements are calculated as follows:

$$C_{ik} = \sum_{j=1}^{q} A_{ij} B_{jk} \tag{3.11}$$

for all i and all k. At this point it is important to explain the mechanics of these equations. In 3.10, we have indicated the sizes of the matrices. On the right-hand side, the adjacent size parameters of the two pairs $(p \times q)$, $(q \times r)$ both have the same value, q. Upon effecting the product, the repeated parameter q disappears, leaving only the pair $p \times r$ on the left-hand side. This mnemonic is very useful.

We now examine equation 3.11, which is actually a set of pr equations, one for each value of i and one for each value of k. Consider one of the equations in the set, where i has a certain value and k has a certain value. The summation index on the right-hand side if j, while i and k remain constant and with the same values they have on the left-hand side. This means that the entire row i of $\mathbf{A}$, and the entire column k of $\mathbf{B}$ are involved in the summation, and all this just for the single element i, j of $\mathbf{C}$. If we think of row i of $\mathbf{A}$ as a vector with q components, and the column k of $\mathbf{B}$ as another vector also with q components, then the calculations implied in 3.11 are the same as in equation 2.30, which describes the dot product of two multidimensional vectors. In summary: element C_{ik} of matrix $\mathbf{C}$ is calculated as the dot product or row i of $\mathbf{A}$ and columnt k of matrix $\mathbf{B}$. This is, again, a very useful mnemonic.

Matrix multiplication can be extended to the case of multiple factors. For example:

$$\underset{p \times s}{\mathbf{D}} = \underset{p \times q}{\mathbf{A}} \; \underset{q \times r}{\mathbf{B}} \; \underset{r \times s}{\mathbf{C}} \tag{3.12}$$

where the sizes of the matrices are indicated underneath their names, the product is done by first multiplying $\mathbf{A}$ and $\mathbf{B}$, and then multiplying the result by $\mathbf{C}$. Note that all adjacent pairs of factors

Chapter 4

Basic C++ Programming

Only experience can make you a good C++ developer. This course will give you the knowledge you need to understand the basics of C++ programming and to begin working with C++ code. C++ is a language, a language that compilers understand, and it requires practice just like any other language.

This course is exceptional in at least one respect. Normally, a C++ course explains the concepts, provides some examples and perhaps some exercises, and leaves the reader right there, without much access to professional code. In this case, however, we have taken advantage of the fact that the professional Vector and Matrix code is part of the eBook. The student is taken seamlessly from the basic concepts explained in the course, to the actual code. The code is extensively documented and cross-references between course material and code are provided. Students are given a unique chance to practice and develop their skills quickly to a level that only developers with years of experience can achieve.

Still, none of this is a substitute for the actual experience of developing code.

If you are an experienced C++ developer, you can skip this entire chapter and go directly to the professional code presented in the following chapters. But if you are not familiar with C++, you must peruse this material first.

4.1 Introduction

C++ programming is done in terms of *objects*. An object associates properties and behavior in a single, inseparable entity, a computational abstraction consisting of data and a description of the behavior of that data. Similar objects are grouped into classes, and a particular object is said to be an instance of its class.

Objects have been used in Science for centuries. In Physics, a solid body, an atom, an electromagnetic wave, are all objects, because they all have properties and behavior. The laws of Physics describe their behavior in terms of their properties. Similarly, in Mathematics, a vector, a matrix,

5.5 All Classes

Class name	Description
ArrayOfCStr	An array of variable length that can hold pointers to objects of type CStr. The array of pointers is owned by the class and is deleted in the destructor. The objects, however, are not.
ArrayOfDoubles	An array of variable length that can hold doubles.
ArrayOfIntegers	An array of variable length that can hold integers.
ArrayOfStr	An array of variable length that can hold pointers to objects of type Str. The array of pointers is owned by the class and is deleted in the destructor. The objects, however, are not.
CStr	A class defining a substring that is part of a larger NULL-terminated string or an Str object.
Formatter	A class that supports `printf` -style formatting for Str and CStr objects.
Matrix	A general rectangular matrix.
Matrix3	A square matrix of numbers of order 3 x 3.
Matrix3X4	A rectangular matrix of numbers of order 3 x 4.
PArray	A parameterized class used as base for all Array classes. It has an array of variable length that can hold pointers to objects of any type. The array of pointers is owned by the class and is deleted in the destructor. The objects, however, are not.
PMatrix	The abstract class used as base for all Matrix classes. It describes a general rectangular matrix of numbers.
PVector	The base class for all Vector classes. It describes a vector in a space with any number of dimensions.
Str	A NULL-terminated array of characters. It is owned by the class and deleted by the destructor when the object is destructed.
UnitVector3	A unit vector in a three-dimensional space.
Vector2	A vector in a two-dimensional space.
Vector3	A vector in a three-dimensional space.
VectorN	A vector in a space with any number of dimensions.

Method Summary	
public: virtual void	Create(int nComps) Creates a PVector object with the specified number of components.
public: virtual void	Create(int nComps, const double * comps) Creates a PVector object with the specified components.
public: virtual double	DotProduct(const PVector & other) Calculates the dot product of this vector and another given vector.
public: virtual int	FindLargestComponent() Returns the index of the component with the largest absolute value.
public: virtual int	FindSmallestComponent() Returns the index of the component with the smallest absolute value.
public: inline const double &	GetComponent(int index) Returns the value of a component of this vector.
public: inline const double *	GetComponents() Returns a pointer to the the components array of constant doubles.
public: void	GetCopyOfComponents(double * copy) Returns by argument copies of the components of this vector in an array.
public: virtual double	GetMyLength() Returns the length or magnitude of this vector.
public: virtual double	GetMyLengthSquared() Returns the square of the length or magnitude of this vector.
public: inline double *	GetNonconstantComponents() Returns a pointer to the components array.
public: inline int	GetNumberComponents() Returns the number of components of this PVector object.
public: virtual bool	IsZero() Returns true if this vector is zero, false otherwise.
public: virtual void	MakeMeZero() Sets this vector to the zero vector.
public: virtual void	MeEqualsDifferenceBetween(const PVector & a, const PVector & b) Sets this vector equal to the difference between two given vectors.
public: virtual void	MeEqualsMeMinus(const PVector & other) Subtracts a given vector from this vector.
public: virtual void	MeEqualsMePlus(const PVector & other) Adds a given vector to this vector.
public: virtual void	MeEqualsMeTimesConstant(double f) Multiplies this vector by a constant.
public: virtual void	MeEqualsMeTimesDiagonalMatrix(const double * diag) Left-multiplies this vector by a given diagonal matrix.
public: virtual void	MeEqualsMinusMe() Changes the sign of this vector.

6.2.2 PVector Constructor Detail

public: PVector::PVector(int nComps, const double * comps)

Components constructor. It sets `m_nComponents` to `nComps` , invokes operator `new` to create array `components` of size `m_nComponents` , and copies `m_nComponents` values from the given array `comps` into array `components` .

Arguments

- `nComps` Number of components.
- `comps` A pointer to a constant array of doubles that contains the values for the components.

public: PVector::PVector(const PVector & other)

Copy constructor. It sets this vector identical to the given vector by copying the value of `m_nComponents` and all element values from `other` to this object.

Arguments

- `other` The given `PVector` object.

public: PVector::PVector(int nComps)

Size constructor. It sets `m_nComponents` to `nComps` and invokes operator `new` to create array `components` of size `m_nComponents` , but leaves the elements of the array uninitialized.

Arguments

- `nComps` Number of components.

public: PVector::PVector()

Default constructor. It sets `components` to `NULL` .

public: virtual PVector::~PVector()

The virtual destructor. It deletes the array pointed at by `components` if this pointer is not `NULL.`

Inherited from Vector3

protected: int m_nComponents;
protected: double * components;
public: double Angle(const Vector3 & q);
public: double AngleDirected(const Vector3 & q, const Vector3 & m);
public: bool Create(const CStr & text);
public: bool Create(const Str & text);
public: bool Create(double x, double y, double z);
public: virtual void Create(int nComps);
public: virtual void Create(int nComps, const double * comps);
public: Vector3 CrossVector(const Vector3 & uu);
public: virtual double DotProduct(const PVector & other)const;
public: double DotVector(const Vector3 & uu);
public: Vector3 DoubleCrossVector(const Vector3 & uu);
public: int FindLargestComponent();
public: int FindSmallestComponent();
public: Vector3 GenerateArbitraryNormal();
public: double GetComponent(int index);
public: void GetComponents(double &ux, double &uy, double &uz);
public: void GetComponents(double * array);
public: inline const double * GetComponents();
public: void GetCopyOfComponents(double * copy)const;
public: double GetMyLength();
public: double GetMyLengthSquared();
public: Vector3 GetMyPNormal(const Vector3 & w);
public: Vector3 GetMyUnit();
public: bool GetMyUnit(Vector3 & unit);
public: inline double * GetNonconstantComponents();
public: inline int GetNumberComponents();
public: Matrix3 GetWMatrix();
public: bool IsZero();
public: double MakeMeUnit();
public: void MakeMeZero();
public: void MeEqualsDifferenceBetween(const Vector3 & a, const Vector3 & b);
public: virtual void MeEqualsDifferenceBetween(const PVector & a, const PVector & b);
public: virtual void MeEqualsMeMinus(const PVector & other);
public: virtual void MeEqualsMePlus(const PVector & other);
public: virtual void MeEqualsMeTimesConstant(double f);
public: virtual void MeEqualsMinusMeMinus(const PVector & other);

Chapter 8

The Array Family of Classes

8.1 All Array Classes

Class name	Description
ArrayOfCStr	An array of variable length that can hold pointers to objects of type CStr. The array of pointers is owned by the class and is deleted in the destructor. The objects, however, are not.
ArrayOfDoubles	An array of variable length that can hold doubles.
ArrayOfIntegers	An array of variable length that can hold integers.
ArrayOfStr	An array of variable length that can hold pointers to objects of type Str. The array of pointers is owned by the class and is deleted in the destructor. The objects, however, are not.
PArray	A parameterized class used as base for all Array classes. It has an array of variable length that can hold pointers to objects of any type. The array of pointers is owned by the class and is deleted in the destructor. The objects, however, are not.

8.2 Class PArray

A parameterized (template) container class representing an array of variable length containing objects or pointers to objects of any type. For types other than primitive types, the default constructor is used to initialize the items in the array, and therefore the existence of a default constructor is required for such types. The class offers support for all standard array operations, including inserting, appending or prepending items or other arrays, sorting, and removing and extracting items or arrays. The single type parameter is called `TYPE` . The class can be used to declare arrays directly, for example:

Index