Appendix B

Software

This appendix describes the software needed for the computer exercises and problems in this book. It has been tested on Windows 7, Linux, and OS X. Please go to the book's web site for any updates to this information.

If you are not able to install the software with the directions below, please find someone to help you.

The three pieces of software you need are Python, sympy, and ga. All are freely downloadable from the web. Python is a multiplatform computer language. SymPy is a computer algebra system written in Python. Commercial computer algebra systems include Maple and Mathematica. These systems provide *symbolic* computation capabilities. ga, also written in Python, adds symbolic geometric algebra and calculus capabilities.

- **Python**. Install the latest Python 2.7 version from http://www.python.org/download/releases/. (Python 3 will not work.) Full documentation is at http://docs.python.org.
- **sympy**. Install the latest Python 2 version from http://code.google.com/p/sympy/downloads/list. Full documentation is at http://docs.sympy.org
- **ga**. Download ga.zip from the book's web page. Unzip it into your site-packages directory. There is a file setgapth.py in the new ga directory. On a Windows system simply run it. On a Linux or OS X system open a terminal in the ga directory and enter "sudo python ./setgapth.py". You will be asked for your password.

  This appendix provides documentation of the ga module for use with this book. It gives only a minimal introduction to ga sufficient to solve problems from the text. In some situations there are simpler approaches to those described here. But to include them would complicate this appendix. Full documentation is in the file ga.pdf in ga.zip.
- **numpy**. This is Python's package for numerical calculations. Install the latest version from http://sourceforge.net/projects/numpy/files/NumPy/. Full documentation is at http://docs.scipy.org/doc/

Vector Calculus

The file "Template.py" at the book's website provides a template for your Python programs. Remove or uncomment lines before starting with your program to suit your needs.

The first line in all SymPy programs should import SymPy:

```
from sympy import *
```

Differentiation, including partial differentiation. Note that everything on a line following a "#" is a comment.

```
x,y = symbols('x y') # Define the symbols you want to use.
print diff(y*x**2, x)
```

Output: `2*x*y`

```
print diff(diff(y*x**2,x),y)
```

Output: `2*x`

Jacobian. Let X be an $m \times 1$ matrix of m variables. Let Y be an $n \times 1$ matrix of functions of the m variables. These define a function $\mathbf{f}\colon X \in \mathbb{R}^m \mapsto Y \in \mathbb{R}^n$. Then `Y.jacobian(X)` is the $n \times m$ matrix of $\mathbf{f}'_{\mathbf{x}}$, the differential of $\mathbf{f}$.

```
r, theta = symbols('r theta')
X = Matrix([r, theta])
Y = Matrix([r*cos(theta), r*sin(theta)])
print Y.jacobian(X)        # Print 2 × 2 Jacobian matrix.
print Y.jacobian(X).det()  # Print Jacobian determinant (only if m = n).
```

Sometimes you want to differentiate `Y` only with respect to some of the variables in `X`, for example when applying Eq. (3.25). Then replace `X` in `Y.jacobian(X)` with only those variables. For example, the command `print Y.jacobian([r])` produces the 2×1 matrix $\left[\begin{smallmatrix}\cos\theta\\ \sin\theta\end{smallmatrix}\right]$.

Integration.
`integrate(f,x)` returns an indefinite integral $\int f\,dx$.
`integrate(f,(x,a,b))` returns the definite integral $\int_a^b f\,dx$.

```
x = Symbol('x')
print integrate(x**2 + x + 1, x)
```

Output: `x**3/3 + x**2/2 + x`

Iterated Integrals.
This code evaluates $\int_{x=0}^{1}\int_{y=0}^{1-x}(x+y)\,dy\,dx$:

```
x, y = symbols('x y')
I1 = integrate(x + y, (y, 0, 1-x))
I2 = integrate(I1, (x, 0, 1))
```

evalf.

```
print log(10), log(10).evalf(3)
```

Output: `log(10) 2.30`

Geometric Calculus

If you want to use the ga module of SymPy, you must import it and specify the $\mathbb{G}^n$ in which you want to work. For $\mathbb{G}^3$ give these commands:

```
from sympy import *
from ga import *
basis = 'e1 e2 e3'
metric = '1 0 0, 0 1 0, 0 0 1'
coords = (x,y,z) = symbols('x y z')
(e1,e2,e3,grad) = MV.setup(basis,metric,coords)
```

Now you can define multivectors, e.g, `M = 3*x*y*e1 + 4*e1*e3`. These operations on multivectors are available:

$+$	add
$-$	subtract
$*$	geometric product
$<$	inner product (as defined in Appendix A)
$\wedge$	outer product
`M.norm()`	$\|M\|$
`M.norm2()`	$\|M\|^2$
`M.grade(k)`	$\langle M \rangle_k$
`M.grade()`	$\langle M \rangle_0$
`M.rev()`	$M^\dagger$
`MV.I`	$\mathbf{I}$ Normalizing can fail. `MV.i` gives unnormalized pseudoscalar.
`MV.Iinv`	$\mathbf{I}^{-1}$

If you see the arithmetic expression $2 + 3 * 4$ you know to multiply $3 * 4$ first and then add 2. This is because mathematics has a *convention* that multiplication comes before addition. We say that multiplication has higher *precedence* than addition. If you want to add first, write $(2 + 3) * 4$.

For the arithmetic symbols used this book, Python's precedences from high to low are $*$, $+-$, $\wedge$, $<$. Plus and minus are grouped because they have the same precedence. This book's precedences are $*$ (geometric product), $\wedge$ (outer product), $<$ (inner product), $+-$. The low precedence of $+-$ causes a problem. Consider the expression $\mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{e}_3$. Python evaluates this as $(\mathbf{e}_1 + \mathbf{e}_2) \cdot \mathbf{e}_3 = \mathbf{0}$. If you intend $\mathbf{e}_1 + (\mathbf{e}_2 \cdot \mathbf{e}_3) = \mathbf{e}_1$, in accord with the precedences of this book, then you must use parentheses.

Here are the rules to get the results you intend. Break a multivector expression into what you consider its terms. Call a term *safe* if it is a scalar times a geometric product of vectors. Most terms are safe or can be rewritten as safe. Fully parenthesize terms which are not safe, including parentheses around them.

Consider again the expression $\mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{e}_3$. If you consider its terms to be $\mathbf{e}_1$ and $\mathbf{e}_2 \cdot \mathbf{e}_3$, then $\mathbf{e}_2 \cdot \mathbf{e}_3$ is not safe and must be rewritten $(\mathbf{e}_2 \cdot \mathbf{e}_3)$. If you consider it to be a single term, $(\mathbf{e}_1 + \mathbf{e}_2) \cdot \mathbf{e}_3$, then it is not safe, and must be written that way. If there are other terms, it should be written $\big((\mathbf{e}_1 + \mathbf{e}_2) \cdot \mathbf{e}_3\big)$.

∇. The variable `grad` returned from `MV.setup` above is a special "vector" which represents the gradient ∇.

`grad` is applied to functions. Functions can be defined in two ways:

```
A = MV('A','vector', fct=True)
A = x + y*e1 + (z*e1 ∧ e2)
```

(How would this evaluate without parentheses?)

The first line creates a *general* vector valued function on $\mathbb{R}^3$ with arbitrary coefficients. The possible grades are `scalar`, `vector`, `bivector`, `spinor` (an even multivector – see Problem 4.2.4), and `mv` (general multivector).

The second line creates a *specific* multivector valued function. (If $\mathbf{e}_1$ and $\mathbf{e}_2$ are orthogonal, then '$\wedge$' can be replaced with '$*$', and the parentheses removed.)

Now we can compute the gradient, divergence, and curl of A:

`grad * A`, `grad < A`, `grad ∧ A`.

Sometimes you want to substitute specific values in a function:

```
(grad ∧ A).subs({x:1,y:2,z:3})
```

Directional Derivative.

```
(h1, h2, h3) = symbols('h1 h2 h3')
h = h1*e1 + h2*e2 + h3*e3
print 'Directional derivative =', DD(h,A)
```

Unfortunately, `(h < grad)*A` will not work.

Curvilinear Coordinates. Curvilinear coordinates must be defined before they can be used:

```
coords = (rho,phi,theta) = symbols('rho phi theta')
curv = \
  [rho*sin(phi)*cos(theta), rho*sin(phi)*sin(theta), rho*cos(phi)],
  [1,rho,rho*sin(phi)]
(erho,ephi,etheta,grad) = \
  MV.setup('e_rho e_phi e_theta', metric, coords, curv=curv)
```

The two single "\" characters designate a line continuation. Python does not require a "\" for the other line continuation. (But it is permitted there.) I used line continuations because of the length of lines in this book. You might not need them in your Python programs. Even if you don't need them, they can improve readability of your code.

The first part of `curv` defines spherical coordinates (Eq. (1.10)). The second part specifies that $|\mathbf{x}_\rho| = 1$, $|\mathbf{x}_\phi| = \rho$, $|\mathbf{x}_\theta| = \rho \sin\phi$ (Exercise 5.23a).

Now proceed with `grad` as before.

Reciprocal Basis. `(xur, xvr, xwr) = ReciprocalFrame((xu,xv,xw))` returns the reciprocal basis of (xu, xv, xw) in (xur, xvr, xwr).

Manifolds

To do geometric calculus on a manifold import a third module:

```
from sympy import *
from ga import *
from manifold import *
```

Then set up a $\mathbb{G}^n$ as described above.

Define a manifold M parameterized by $\mathbf{x}(u, v)$:

```
Mvar = (u,v) = symbols('u v')
X = u*e1 + v*e2 + (u**2+v**2)*e3
M = Manifold(X,Mvar)
(xu,xv) = M.basis
```

There are two ways to define a field $\mathbf{f}$ on M:

```
f = (v+1)*xu + u**2*xv    # Use the tangent space basis.
f = u*e1 + v*e2 + u*v*e3  # Use the R^n basis.
```

Compute the vector derivative $\boldsymbol{\partial}\mathbf{f}$, divergence $\boldsymbol{\partial}\cdot\mathbf{f}$, curl $\boldsymbol{\partial}\wedge\mathbf{f}$:

```
M.grad * f,  M.grad < f,  M.grad ∧ f.
```

Compute the directional derivative $\partial_{\mathbf{h}}(\mathbf{f})$:

```
h = xu + xv
M.DD(h,f).
```

Project a field $\mathbf{f}$ defined on M to M:

```
M.Proj(f).
```

Use this to compute a coderivative (Problem 5.5.1).

Printing

By convention, a single underscore "_" in output indicates a subscript and a double underscore "__" indicates a superscript.

```
(e1,e2,e3) = MV.setup('e_1 e_2 e_3', metric)
A = MV('A','vector')
print 'A =', A
```

Output: `A = A__1*e_1 + A__2*e_2 + A__3*e_3`

The coefficients are superscripted by convention. Notice that `e1` is used in the left side of the first line and `e_1` on the right. `e1` is the variable name used in the program. `e_1` is what is printed.

There are three options for improving the looks of the output of `print` statements: (i) Fmt, (ii) enhanced printing for console output, and (iii) LaTeX output compiled to a pdf file.

Fmt. The command `Fmt(n)` specifies how multivectors are split over lines when printing:

$n = 1$: The entire multivector is printed on one line. (The default.)

$n = 2$: Each grade of the multivector is printed on one line.

$n = 3$: Each component of the multivector is printed on one line

The $n = 2$ and $n = 3$ options are useful when a multivector will not fit on one line. If the code `A.Fmt(n)` is executed, then `A` will print as specified. If `A.Fmt(n,'A')` is executed, the string '`A =` ' will print, followed by `A`, as specified by `n`. You can print a variable with one n and later with another.

Enhanced printing. If you are sending your output to a console window, i.e. you are not using the LaTeX output described below, then you may issue the statement

```
enhance_print()
```

after the `MV.setup` statement. Then bases are printed in blue, functions in red, and derivative operators in cyan, making the output more readable.

To set this up in Windows you must tell Geany to use a special console program in GA.zip. Go to Edit/Preferences/Tools/Terminal. Navigate to and select

Lib\site-packages\GA\dependencies\x32\ansicon.exe or

Lib\site-packages\GA\dependencies\x64\ansicon.exe,

according as you are running a 32- or 64-bit system.

On Linux or a Mac you need do nothing.

LaTeX output. If you have an appropriate LaTeX system on your computer,[1] then output from Python print statements can be sent to a tex file and automatically compiled and displayed on a pdf reader, with beautiful LaTeX typesetting. It is helpful, but not necessary, to know a bit of LaTeX for this.

Invoking LaTeX printing requires an extra import `from ga_print import *`, `Format()` after your import statements, and `xdvi()` at the end of your program. On Windows, the pdf output is opened in the associated pdf reader. On Linux, it is opened in the standard *evince* pdf reader.

Here is an example of this capability. When printing a string, an underscore "_" designates a subscript. A caret "^" (not a double underscore) designates a superscript.
`print '\\alpha_1\\bm{X}/ \\gamma_r^3'`
Output: $\alpha_1 \boldsymbol{X}/\gamma_r^3$.

The file Symbols.pdf, available at the book's website, is a listing of common LaTeX symbols. The symbols there are preceded with a "\". They can be printed from your Python programs by preceding them with a second "\", as in the example.

This example prints some geometric algebra/calculus symbols:
`print '\\bm\\nabla, \\wedge, \\cdot' \\partial, \\bm\\partial'`
Output: ∇, $\wedge$, $\cdot$, ∂, $\boldsymbol{\partial}$

In LaTeX mode, the statement `print 'A =', A` from the beginning of this heading produces the output $A = A^1\mathbf{e}_1 + A^2\mathbf{e}_2 + A^3\mathbf{e}_3$. The coefficients are superscripted by convention.

This extends:
`(ax, bx) = symbols('a_x, b_x')`. Then `ax` and `bx` print with subscripts.
`(ax, bx) = symbols('a__x, b__x')`. Then `ax` and `bx` print with superscripts.

[1] TeX Live is known to work, as is MiKTeX on Windows.