

Math/CS 365, Spring 2004

## Scientific Computation

Daniel Kaplan, Macalester College

### Project I: Machine Arithmetic

Reliable computer arithmetic is now widely available, including the sophisticated double-precision hardware following the IEEE 754-1985 standard and arbitrary precision capabilities of packages such as Mathematica. And yet it's still a mistake for users to think that the entities being manipulated are mathematical numbers. They may behave a lot like numbers sometimes, but other times they don't and the unaware user can get seriously burned.

To help you understand how it's possible to build a software arithmetic system that works like numbers, and how it fails sometimes to work like numbers, you are going to implement a rudimentary integer and floating-point system. Our reasons for doing this are purely pedagogical; technically superior systems are easily available. But when you have finished, in addition to having a much better understanding of the basic algorithms for arithmetic, you will have a hard-earned visceral appreciation of both the intricacies and limitations of machine arithmetic.

Your job:

1. Write operators for signed integer addition, subtraction, and multiplication of computer numbers of any integer base and any number of digits.
2. Building on your integer operators, write a system of floating point operators for addition, subtraction, multiplication, and division.
3. Using a system of types and operators that I will provide for floating point arithmetic (or, using your own system if you like), implement a variety of calculations such as  $\exp x$ ,  $\ln x$ ,  $\sin x$ , polynomial evaluation, and so on.

This project involves much more writing of software than any of the future projects. If you don't already know MATLAB, it is hoped that this will get you off to a good start in learning.

Your project report should consist of two parts:

1. The programs you write, sent to me via email as a zip-format (or, on Linux, a tar-format) folder.
2. An **annotated** MATLAB dialog demonstrating and exercising the operations.

#### Getting the Software

There are several MATLAB m-files that you will use for this project. These are located on the `academic/groups/common/mathcs/math-cs-365` course folder in a directory named `arithmetic`. (A zip-compressed copy of this directory is available via the course web site.) To use these m-files, copy them to the computer you are working on and set the MATLAB path to refer to this directory and to its subdirectories.

Note that many of the files are contained in directories starting with the character `@`. Such directories are part of MATLAB's object-management system. While you don't need to know how this system works to use the software, you should make sure to keep the files in those directories.

#### Extending the Project

As we go through the semester and you think about choosing your term project, keep in mind some ways that this project could be extended:

- Write a numerical system for interval arithmetic, where instead of operating on single numbers we operate on pairs of numbers reflecting the possible range of a computed value. This interval needs to be automatically propagated through all computations.
- Write a system for "automatic differentiation." In symbolic differentiation, as you know, recursive rules of symbolic manipulation are used to find an analytic form for the derivative of a function. In numerical differentiation — a topic that will be covered later in the semester — we compute the derivative of a function  $f(x)$  by computing a finite difference, something like  $\frac{f(x+h)-f(x)}{h}$  for  $h$  "small." Automatic differentiation is between these two styles. By adopting special rules for arithmetic operators on pairs of numbers, a function's derivative can be computed at the same time as the function itself. This effectively allows us to compute the numerical derivative as if we had substituted in a value into an analytic derivative, but without needing to compute explicitly the analytic form.

## Integer Arithmetic

You will first need to decide how you want to represent the information that is contained in an integer. This information is: the sign, the digits, the base. In order to save you trouble, and to get people off to a fast start who don't already know MATLAB, I have written a simple way to store this information and to translate to and from the native numerical type (which is a double-precision floating point).

The program `makeInt` takes a native number as an argument, and a base, and produces a MATLAB structure that holds the base, sign, and digit information. Here is 225 in base 10:

```
>> a = makeInt(225, 10)
a =
    b: 10
    s: 1
    d: [5 2 2]
```

Note that the digits are stored as an ordinary MATLAB vector. The least significant digit is, contrary to conventional notation, to the left. This is arbitrary, but it highlights the fact that numbers are just data and you can arrange those data in any way that's convenient.

You can use any integer base that you want, e.g.,

```
>> a = makeInt(225, 3)
a =
    b: 3
    s: 1
    d: [0 0 1 2 2]
```

Another operator, `toDoubleInt`, converts from the structure information to the native numerical type. For instance,

```
>> toDoubleInt(a)
ans =
    225
```

You can access the fields of the structure using the dot operator, e.g.,

```
>> a.d
ans =
    0    0    1    2    2
```

Here is your task in detail:

1. Start by writing an operator `addInt(x,y)` that takes two positive integers (in this structure format) as input and returns their sum. You can use the conventional grade school algorithm. The two arguments should have the same base. If the sum has an extra digit, include it in the result.

2. Write a `negateInt` operator. All this needs to do is change the `s` field of the integer.
3. Write a `subtractInt(x,y)` operator that takes two positive integers and computes their difference. In order to accomplish this, you may want to implement a "radix-complement" operator (the generalization of twos-complement)<sup>1</sup>, then just add `x` to the radix complement of `y`. If the result is negative, complement it back and adjust the sign field appropriately.
4. Modify `addInt` and `subtractInt` so that they handle negative and positive arguments appropriately.
5. Write a `singleDigitMultiplyInt(x,digit)` operator that multiplies `x` by a single digit less than the base. This can be done using a multiply-and-carry method similar to that in addition.
6. Write a `multiplyInt(x,y)` operator that multiplies `x` and `y`. This can be done by multiplying `x` by each digit of `y` in turn, and adding up the results with appropriate shifting of digits to account for the place of the digit in `y`.

## Floating-point Arithmetic

A floating-point number consists of a mantissa — an integer, which has a sign, digits and a base — as well as an exponent. The floating point operations can make use of the integer operations you have already programmed.

1. Write a `makeFloat(num,base,exponent)` operator that works analogously to `makeInt` but returns a structure with an `exp` field. You can represent the exponent as a native number, which allows for exponents that are hugely greater than those that will be encountered in practice.
2. Write `toDoubleFloat(x)` that returns the value of `x` as a native type.
3. Floating point multiplication is the simplest operation to implement. Write `multiplyFloat(x,y)` which returns the product of `x` and `y`. Think about how you need to combine the two mantissa and the two exponents to produce the result. It's not terribly hard.
4. Write an operator `addFloat(x,y)`. In general, you will need to shift the digits of one of `x` or `y` before performing an integer add. Arrange to trim the result to the number of digits in `x` or `y`. You can either truncate or round, but do so properly.
5. Similarly, write `subtractFloat(x,y)`.

<sup>1</sup>Algorithm: subtract each digit from the base minus 1. Then add 1 to the resulting number.

<sup>2</sup>Starting with an excellent approximation to  $y = 1/x$  (perhaps using the native floating-point arithmetic), follow the recursion  $y_{k+1} = y_k(2 - xy_k)$ .

If you have gotten your floating point system to work, you may want to add a `recipFloat(x)` operator. This can be done using Newton's method.<sup>2</sup> With `recip`, you will be in a position to write `divideFloat(x,y)` using your already developed multiplication, addition, and subtraction operators.

## Building on the Floating-point Numbers

It's challenging to program reliable floating-point numbers. Integrating them as a new type into MATLAB requires additional programming knowledge that isn't covered in CS 121. For this reason, I'm providing you with a ready-made floating point system. (The system has been extensively tested on random numbers, but it isn't perfect. Please report bugs to me, giving me both the operator and the arguments that caused the failure.)

The `fp(val, base, ndigits)` operator constructs a floating-point number. Here, for example, we take advantage of the fact that the MATLAB variable `pi` has a finite-precision approximation to  $\pi$ :

```
>> fp(pi, 10, 5)
[ + 3 . 1 4 1 6] x 10^0
```

Note that the number is printed out in something like the conventional format. We've asked for 5 digits of  $\pi$  in base-10.

Here's the same thing, but in base-100. Each of the "digits" is actually a number between 0 and the base minus 1. You can read off the first 9 decimal digits of  $\pi$ .

```
>> fp(pi, 100, 5)
[ + 3 . 14 15 92 65] x 100^0
```

Using a base other than 10 gives an unfamiliar output, but the value is correct:

```
>> fp(pi, 107, 5)
[ + 3 . 15 16 10 10] x 107^0
```

The `fp` system has been written so that you can use the conventional infix notation for addition, subtraction, multiplication, and division. (NB. The notation for multiplication is `.*` and for division is `./` — don't forget the dots!) Here's  $1/3$  computed using different bases, carrying each out to 9 digits:

```
>> 1./fp(3,10,9)
[ + 3 . 3 3 3 3 3 3 3 3] x 10^-1
```

Base-10 gives the conventional 0.33333333, a repeating decimal.

Base-2 is also a repeating decimal

```
>> 1./fp(3,2,9)
[ + 1 . 0 1 0 1 0 1 0 1] x 2^-2
```

The representation is exact in base-3

```
>> 1./fp(3,3,9)
[ + 1 . 0 0 0 0 0 0 0 0] x 3^-1
```

All that's happening is that we are writing  $\frac{1}{3}$  as  $3^{-1}$  — a symbol game.

1. Write a program `mycos(x)` to compute  $\cos x$ .
2. Write a program `mypoly(a,x)` to compute the degree- $n$  polynomial  $a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ . The argument `a` is a vector containing the  $n + 1$  coefficients  $a_i$ .
3. Write a program `myexp(x)` to compute  $\exp x$ .
4. Write a program `mycos(x)` to compute  $\ln x$ .
5. Write a program `sroot(x)` that computes the square root of a `fp` number using Newton's method.