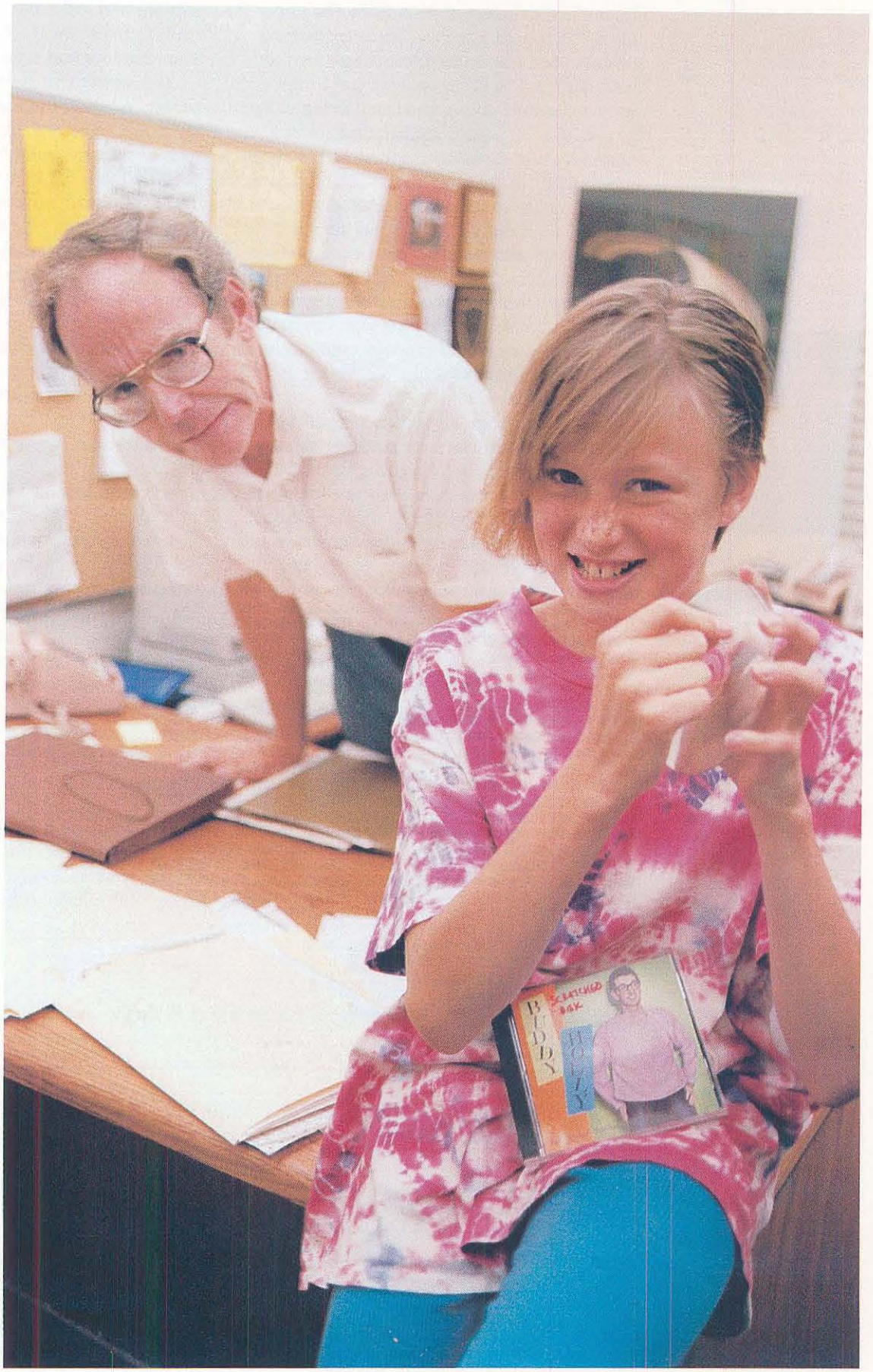

BUDS
SCRATCHED
BUDS

Safety in Numbers: Protecting Data Mathemagically

by Robert J. McEliece

Lizzy McEliece scratches a favorite compact disk with a paper clip as her father winces in mock horror—although he knows that error correction will save his music. Unfortunately, this time excessive scratching zeal finally did in this CD, which had, however, performed beautifully after a similar demonstration at the Watson Lecture.

They say that people with emotional problems become psychologists, sick children grow up to become doctors, people with bad teeth become dentists, and so on. Well, when I was little, I made a lot of careless mistakes: I could not read my own handwriting; I made spelling mistakes; I forgot to carry when I multiplied; when I tried to play chess, I always overlooked something. But after my junior year at Caltech, I got a summer job at JPL, where I found out that there was a way to use mathematics to correct errors automatically! For some reason, this subject fascinated me immediately. I learned all about it, and, as it turned out, the theory of error correction became a big part of my life's work. I still make a lot of errors, but now I get paid for correcting them. (And I get paid very nicely, thank you.) So, in this article I'd like to explain some things about the theory of error correction—why I think it's a lot of fun, and how it plays a big part in the design of a lot of today's communication and data-storage systems.

Everyone is already pretty good at error correction, in English at least. Below is a five-letter English word, in which one of the letters has been erased and replaced with a question mark:

Q?EEN

That's not so hard, is it? Okay, here's a slightly harder one—a five-letter word in which there's an error. One of the letters has been changed to another:

TRADF

Well, maybe that wasn't too hard either. But you should congratulate yourself anyway, because a computer has a much harder time than you do solving problems like this. In fact, no one knows exactly how humans correct errors in English. As a rule, though, it's about twice as hard to correct an error as an erasure.

Here are five more words to try. Change one letter and make a common English word. (Don't spend too much time on this or you'll never get on with this story. The answers are on page 36.)

THARA
ADAST
TRENA
NEMVE
SPLIR

If you got a couple or all of them, you can see that error correction in English is possible, even if it's not always so easy. It's possible because there is a subtle and complex pattern to the way words and sentences are constructed in English. Everybody knows that Q is always followed by U. That's a pattern, and even if the pattern is partially destroyed (by erasing the U, for example), the word is still recognizable. Many words end with the pattern: *vowel, consonant, silent E.* Such words as *date, stalactite, remote,* or *cute*—or *trade*—illustrate this pattern. And even if the silent E is changed to F or something, it's still pretty easy to see the pattern and tell what the word is. In the last five words, however, the pattern isn't as strong, and that's why it's harder.

Sometimes the patterns in English aren't good enough. Here's another five-letter word with one erased letter:

S?OUT

There is no one right answer to this one, because there are at least five possible answers (shown at left).

SCOUT
SHOUT
SNOUT
SPOUT
STOUT

The problem is that these five words are too close to each other; they differ from each other by only one letter, and if the second letter is erased, the word is lost. You can probably think of lots of other examples. Can you find a word of five or more letters such that if you erase one letter, there are six or more possible completions? (One suggestion appears on page 36.) With errors instead of erasures, the situation can be even more complex. Here's a five-letter word in which one of the letters has been changed:

GLADE

BLADE
GRADE
GLIDE
GLARE
GLADS

Well, GLADE is already a word, and there are (at least) five more words (shown at left) that we can get from it by changing just one letter.

You can see that GLADE is rather sensitive to possible typos. In fact, in 1879 Lewis Carroll invented a word game, called "doublets," based on the fact that many pairs of words differ by only one letter. For example, we can change BLACK *into* WHITE by making a sequence of one-letter changes. If you want to play "doublets," you might try turning LEAD into GOLD. (See page 36.)

BLACK
SLACK
STACK
STALK
STALE
SHALE
WHALE
WHILE
WHITE

So English has quite a lot of built-in error-correcting ability because of its natural patterns. But it wasn't designed systematically to correct errors, and, as we have seen, sometimes changing just one letter in a word can dramatically change its meaning. Of course, it is just this wonderful ambiguity that lets us play word games, commit horrible puns, write complex poetry, and invent pseudo-words like "mathemagically." I don't advocate changing the English language.

But suppose we did want to design a very precise language for absolutely reliable communication of important information (air-traffic control, military commands, deep-space communication, and so on). It turns out that we can do much better than English—no fun with word games, no clever poetry, but a much better ability to recover from errors.

The first thing to know, if you want to design a new and improved language for communicating reliably despite errors, is that you don't have to use a 26-letter alphabet. (It's like Wilbur and Orville Wright designing the airplane: they looked at the birds for ideas, but they didn't have to copy slavishly. Birds have wings; that's a good idea. The wings flap; that's not such a good idea. Airplanes have things that spin around on their noses; they don't have feathers, and so on. We can steal ideas from nature if we want to, but only if we want to.) In fact, an alphabet with only two letters *is* sufficient, and, as you may know, many modern communication and data-storage systems use a two-letter alphabet. That's what a digital communication system is, really—a two-letter system.

The letters in these two-letter alphabets are usually called *zero* and *one*, but these names are arbitrary, and any other pair of names would do as well, for example, *yes/no, up/down, on/off, high/low,* or *trivial* and *obvious* (Caltech students' favorite two words). However, it's not trivial or obvious that with an alphabet of only two letters you can communicate any possible message. In fact, there is a famous episode of "Star Trek" that hinges on just this point.

This episode, called "The Menagerie," tells the story of the unfortunate Captain Christopher Pike, who has been exposed to a near-fatal dose of delta rays and is confined to a sort of tin can. He is able to see and hear and think normally, but he cannot speak. His tin can, however, has a light on the front that Captain Pike can use to communicate—one flash for yes and two for no. He has a two-letter alphabet. Captain Kirk tries for hours to figure out what's wrong with poor Captain Pike, who seems to be upset about something. He is joined by Dr. McCoy, who is not much help, though he gets quite philosophical about it. The stardate is 3012.4, and the dialog runs as follows:

Capt. Kirk: He keeps blinking "no"—no to what?

Dr. McCoy: They've tried questioning him. He's almost agitating himself into a coma.

Capt. Kirk: How long will he live?

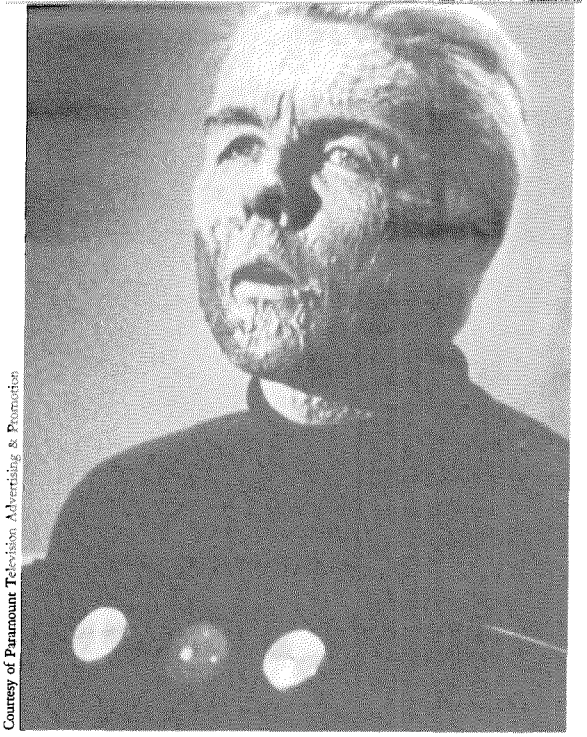
Dr. McCoy: As long as any of us. Blast medicine anyway; we've learned to tie into every human organ in the body except one—the brain. The brain is what life is all about. Now, that man can think any thought that we can, and love, hope, dream as much as we can. But he can't reach out and no one can reach in.

Capt. Kirk: He keeps blinking "no"—no to what?

Dr. McCoy: They can question him for days, weeks, before they stumble on the right thing.

Capt. Kirk: Could this have anything to do with Spock?

Yes, of course it has something to do with Spock. What's wrong with these guys? Captain Pike's mind isn't really trapped in there! They should study digital communications. There are many ways they could find out what's wrong. For example, they could have said to Captain Pike, "Think of a sentence describing your problem. Is the first letter A?" and so on, and in five



Courtesy of Paramount Television Advertising & Promotion

In a vintage Star Trek episode poor Captain Pike was exposed to a near-fatal dose of delta rays and locked up incommunicado except for a light that can flash "yes" or "no." He should have remembered how to play 20 questions.

But even the logical Mr. Spock may not have known that if Captain Pike's little light had malfunctioned occasionally . . . it would still have been possible to communicate reliably with him.

minutes they could have found out that Spock was planning to kidnap Captain Pike and take him to the forbidden planet Talos IV. In fact, any conversation that could be carried out between ordinary folks could also be carried out with Captain Pike, with a little patience and ingenuity. If you've ever played 20 questions you already know this.

In the game of 20 questions there are two players. Player 1 thinks of some object, and Player 2 tries to guess what the object is by asking Player 1 a series of questions that can be answered *yes* or *no*. In the traditional version of the game, Player 2 is allowed to ask up to 20 questions. As a simplified illustration, suppose I select one of the letters A, B, C, D, E, F, G, or H, and ask you to try to guess it by asking a series of *yes/no* questions. Our dialog might proceed as follows:

Question: Is it A, B, C, or D?	*Answer*: NO
Question: Is it A, B, E, or F?	*Answer*: YES
Question: Is it A, C, E, or G?	*Answer*: NO

After these three questions you will know what the letter is (in this case, it's F), since every one of the eight possible patterns of *yes/no* answers corresponds to exactly one of the eight letters. (The correspondence is listed on page 36.)

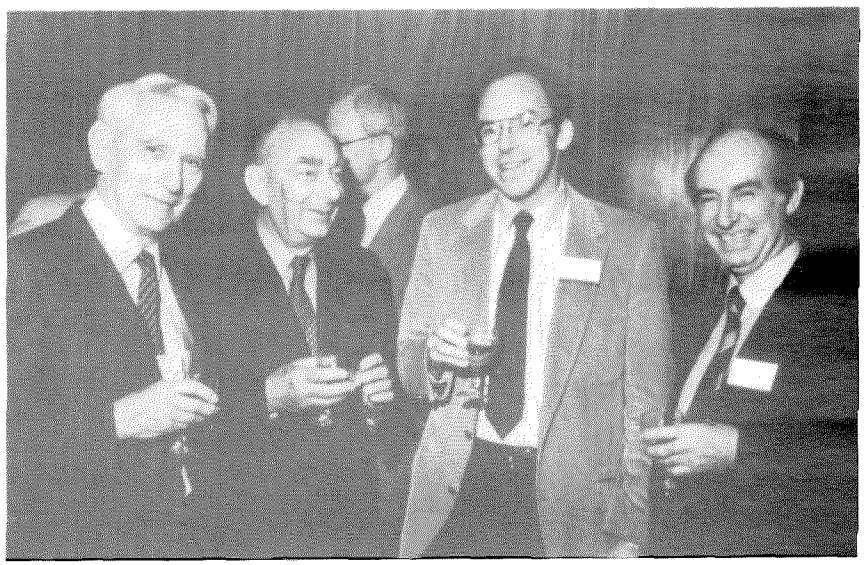
So Captain Kirk could have communicated reliably with Captain Pike by playing 20 questions. But even the logical Mr. Spock may not have known that if Captain Pike's little light had malfunctioned occasionally (by flashing *yes* when he meant *no* or vice-versa), it would *still* have been possible to communicate reliably with him. This fact is far from trivial and obvious, but it's nevertheless true. It can be done by playing "20 questions with lies," a game invented in the 1964 PhD thesis of Elwyn Berlekamp at MIT.

To illustrate how 20 questions with lies works, suppose I think of one of the above eight letters again, and you try to guess it asking *yes/no* questions. This time, however, I am not required to answer truthfully; I can lie sometimes. This time I'll allow you to ask nine questions, and in return you must allow me to lie up to twice in my nine answers. Now our dialog might go as follows:

Question 1: Is it A, B, C, or D?	*Answer:* NO
Question 2: Is it A, B, E, or F?	*Answer:* YES
Question 3: Is it A, B, C, or E?	*Answer:* NO
Question 4: Is it A, E, or F?	*Answer:* YES
Question 5: Is it A or F?	*Answer:* NO
Question 6: Is it F or G?	*Answer:* YES
Question 7: Is it F or G?	*Answer:* YES
Question 8: Is it F?	*Answer:* NO
Question 9: Is it F?	*Answer:* NO

After these nine questions, even though I may have lied to you twice, you will know for sure what the letter is, and which answers were lies. (The solution appears on page 36.) Indeed, Berlekamp's results imply that it is always possible to determine one of eight possibilities with no more than nine questions in the presence of two lies, although the details of the questioning strategy are a little complicated.

The game of 20 questions with lies makes a nice parlor trick, but it also illustrates an important fact: it is possible to communicate reliably even though the communication medium itself is unreliable. This fact, and its remarkable consequences, was discovered in 1948 by a young



Members of the Claude Shannon fan club pose with their idol (left) at the International Information Theory Symposium in Brighton, England, in 1985. McEliece is second from right. At right is Paddy Farrell of the University of Manchester (co-chairman, with McEliece, of the symposium). Shannon showed that any communication process (left) can be rendered reliable by adding redundancy (on opposite page) that creates a strong pattern with the original data.

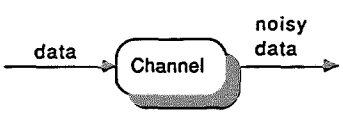
mathematician named Claude Shannon, one of the finest minds of this or any other century. (Berlekamp was a student of Shannon's.) Before I describe his scientific accomplishments as the inventor of information theory, let me tell two Shannon stories.

In 1985 the Japanese government decided to institute a prize for scientific and humanistic achievement, called the Kyoto Prize, which the Japanese hoped would rival the Nobel Prizes. (In monetary value a Kyoto Prize is worth slightly more than a Nobel.) The first Kyoto Prize in Basic Sciences was given to Claude Shannon. I expect there was very little trouble deciding whom to give it to. He would have won a Nobel Prize years ago, except that his achievements are in engineering and mathematics, and there aren't any Nobel Prizes in those subjects. The Kyoto Prizes cover all of science, so he was immediately eligible.

The second story is more personal. Shannon's main work was all done before 1965. Since then he has been semiretired, and a whole generation of researchers (including me) had never met him until June of 1985, when he unexpectedly showed up in Brighton, England, at an International Information Theory Symposium. All of us Shannon fans (which included everyone at the symposium, I can assure you) were thrilled to see him, and cameras were clicking all week. At the closing banquet, Shannon was, of course, seated at the head table. About halfway through the banquet, Lee Davisson, who was at that time head of the electrical engineering department at the University of Maryland, did what we had all secretly wanted to do all week: he asked Shannon for his autograph. That opened the floodgates. For the rest of the meal, there was a long line of autograph hounds (including me) waiting for Shannon's autograph. If you know how large scientific egos tend to be, you'll understand how really astonishing this scene was. It was as if Newton had showed up at a physics conference.

That's enough hagiography. What exactly did Shannon do?

Claude Shannon has a great feeling for generalities. He saw that any communication process—talking to another person either face-to-face or on the phone, watching TV, sending photographs of Neptune to Earth—can be modeled by the simple picture at left. The information that must be communicated is transmitted over a channel—the air that separates two people conversing, a telephone wire, the complicated stuff between the television studio and your house, or the 2.8 billion miles of empty space between Neptune and Earth. All communication channels are to a greater or lesser degree "noisy," which means that what comes out of the channel isn't always exactly what goes in. On many channels, the noise is intolerable—think of a bad phone connection or trying to talk to someone near the airport when a 747 flies over. Until Shannon, everybody thought that the only way to communicate reliably over an unreliable channel was to physically make the channel more reliable—yelling to overcome the 747 noise, or, more generally, building more powerful transmitters or more sensitive receivers, and so on. In



1948, however, Shannon showed that this wasn't necessary. It is, in fact, possible to communicate perfectly reliably over essentially any channel, however noisy it may be. I already showed you an example of this: I was able to communicate to you one letter of the alphabet over a channel that caused errors (lies).

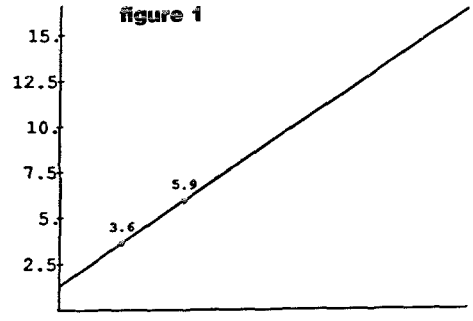
Shannon showed that every channel has an ultimate capability to transmit information . . . and that this limit can be reached only if the channel is making lots of errors.

The illustration above shows Shannon's solution for communicating reliably over an unreliable channel. The idea is to send the data over the channel as shown in the previous diagram; but before the data is sent over the channel, it's processed by a man-made device called an encoder. The job of the encoder is to take the data and use it to calculate something called redundancy. The redundancy is then combined with the data before it's transmitted. Roughly speaking, the redundancy is added so that when the data and redundancy are combined, a strong pattern appears. It's a little like adding a U after every Q. Then the data and the corresponding redundancy (the combination is usually called a codeword) are sent over the noisy channel. Of course, the channel may cause errors in the redundancy as well as in the data. Shannon showed, however, that if the redundancy is computed in just the right way, the resulting pattern in the codeword will be so strong that it will almost always still be recognizable despite the channel noise. The pattern-recognizing device is called the decoder, and its job is to reconstruct the original data, using the noisy data and the noisy redundancy as clues.

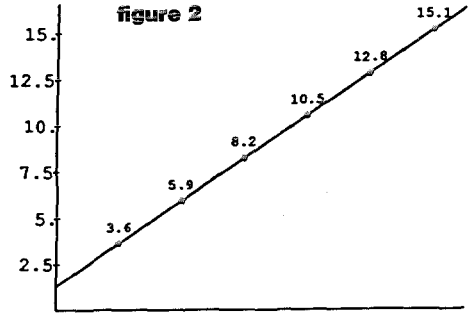
There's a corollary to this, which is perhaps the most important thing about Shannon's work and which is not fully appreciated, even by many professionals today. It's that Shannon's theory applies to all channels, even ones that *aren't noisy*. In fact, Shannon tells us that if the channel isn't making a lot of errors, you're not using the channel to its fullest capacity. For example, if you're communicating photographs from Neptune to Earth, say at the rate of one picture per second, and everything *seems* to be going as well as it could, you're fooling yourself. You should be pushing the channel harder, maybe 10 pictures per second, right to the ragged edge of failure, forcing the channel to make lots of errors, and then correcting the errors, using redundancy and pattern recognition. In this way Shannon showed that every channel has an ultimate capability to transmit information, called the channel's *capacity* or Shannon limit, and that this limit can be reached only if the channel is making lots of errors, which are being corrected by the decoder like crazy. Shannon proved that channel capacity is like the speed of light: with a lot of work (building fancy encoders and decoders) you can get as close to it as you like, but you can never quite get there.

What Shannon did not do, however, was to say exactly how the encoders and decoders should be designed; he only showed that it must be possible. The actual design task he left for later generations, who have accepted that challenge. (Thank goodness he left something for us to do!) Today there are dozens of different error-correction systems in practical use in a wide variety of applications. I can't begin to describe even a few of these error-correction systems in the space of this article, so I'll content myself with just one, which was invented in 1960 by two MIT researchers, Irv Reed and Gus Solomon. Reed is now professor of electrical engineering at USC, and Solomon is a senior scientist at Hughes Aircraft Company (and also a well-known teacher of integrated voice and movement).

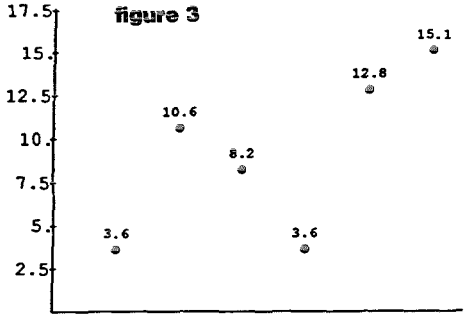
Reed-Solomon codes, as they're now called, began as only a theoretical curiosity, but today they're probably the most widespread and generally useful error-correcting code system. Reed-Solomon codes work, ultimately, with zeros and ones (bits), but it's easier to understand what's going on if you think not in terms of bits but of *bytes*—a group of eight bits. There are 256 possible bytes, numbered from 0 to 255. Reed-Solomon codes protect data that has been grouped into bytes; the data is in bytes, and the redundancy is in bytes, but since the bytes are in correspondence with the numbers 0 through 255, let's just pretend that they work on ordinary numbers.

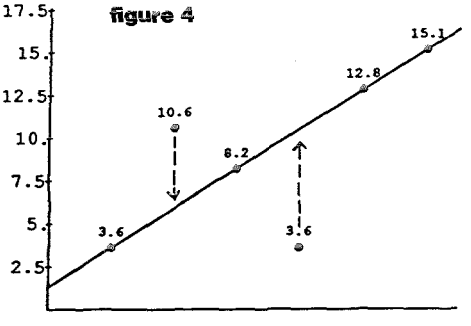


So now let's see how Dr. Reed and Dr. Solomon filled Shannon's prescription. We shall see that the basic ideas are geometric: the information to be transmitted is encoded as part of a strong geometric pattern that can be recognized even if it is partly garbled by the channel.

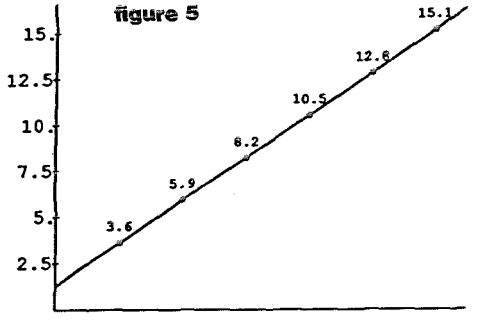


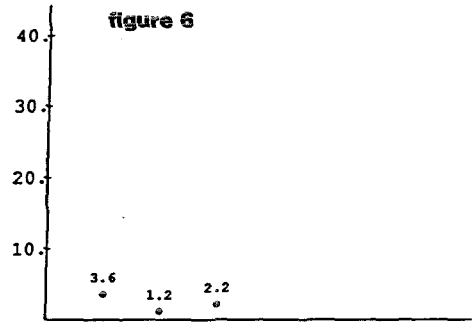
Let's suppose, for example, that we want to transmit just two numbers, say 3.6 and 5.9, from one point to another over a noisy channel, and that we want to *encode* these numbers before transmitting them, using Reed and Solomon's ideas. To do this, we first plot the two numbers geometrically, and then join them with a straight line (figure 1). This straight line began with only two points, but of course now there are lots of other points on it. Let's pick four more of these points, spaced equally along the line, which as you can see from the figure are 8.2, 10.5, 12.8, and 15.1 (figure 2). We then use these four extra numbers as the redundancy and transmit the data plus the redundancy as the codeword [3.6, 5.9, 8.2, 10.5, 12.8, 15.1]. This codeword has a very strong pattern: each number is exactly 2.3 more than the previous one.

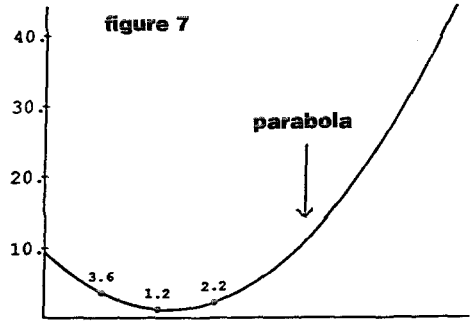


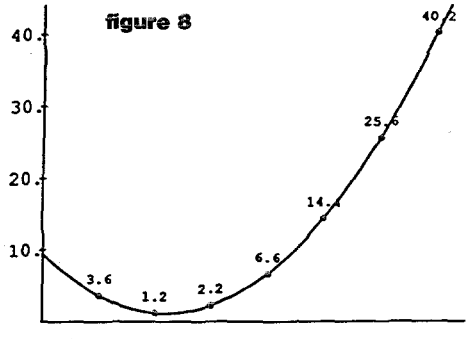


Now let's send our codeword over the channel, and let's say that two errors occur, in the second and fourth positions, so that [3.6, 10.6, 8.2, 3.6, 12.8, 15.1] is received (figure 3). You can see that the straight-line pattern has been spoiled, but not entirely obliterated, since four of the six points still lie on a straight line. If we draw a straight line through these four points, we see that two of the points aren't on the line (figure 4). But since we know that the transmitted points all began on a straight line, in order to recover from the errors, all we have to do is move the wayward points back onto the line (moving vertically, as shown), and presto! the original codeword [3.6, 5.9, 8.2, 10.5, 12.8, 15.1] appears (figure 5). Now the original information [3.6 and 5.9] can be read off, and we have communicated these numbers reliably despite the channel noise.









Although this was just one example, the same process will always work. If the six-number codeword is subjected to two or fewer errors, no matter where the errors occur, at least four of the received numbers will still lie on a straight line, and if the decoder draws that straight line, the erroneous numbers can be moved onto the line and corrected. If the channel is in an especially bad mood, however, and three or more errors occur, the system can fail. Can you see why? (The answer is on page 36.) If you want your codeword to correct three errors, you'll need six redundant numbers; and to correct four errors, you'll need eight extra numbers, and so on.

I've just explained how Reed-Solomon coding is used to protect pairs of numbers: you connect the two points with a straight line, add extra points on the line, etc. In practice, of course, you might want to send three or more numbers at once. How do Reed-Solomon codes do this? To see how, let's consider another example. Suppose we wanted to send the three numbers [3.6, 1.2, 2.2] over a noisy channel. We again plot the three points geometrically (figure 6). Unfortunately, these three points don't lie on a straight line. But any three points will determine a *parabola*, which is a second-degree curve, so let's draw a parabola through these points (figure 7). Now that we have the parabola, it's easy to guess what to do next. We locate some more, say four more, equally spaced points (equally spaced horizontally) on the parabola, and use these points as the redundancy, thereby producing the codeword [3.6, 1.2, 2.2, 6.6, 14.4, 25.6, 40.2] (figure 8). The original data had no particular pattern, but this seven-number codeword has a very strong pattern, because the seven points all lie on the same parabola. (It's extremely unlikely that seven numbers chosen at random would lie on a parabola.)

figure 9

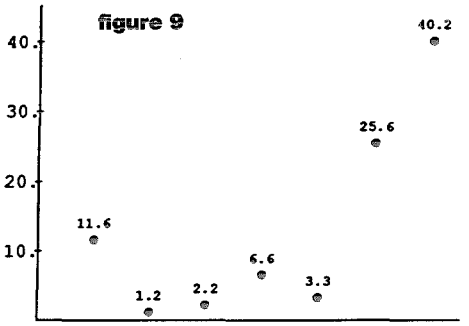


figure 10

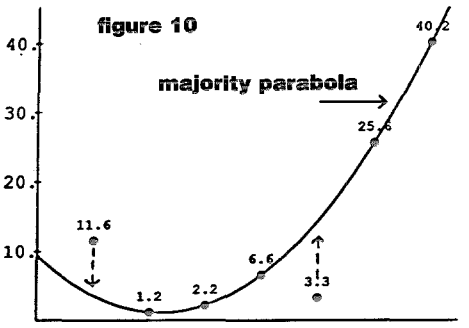


figure 11

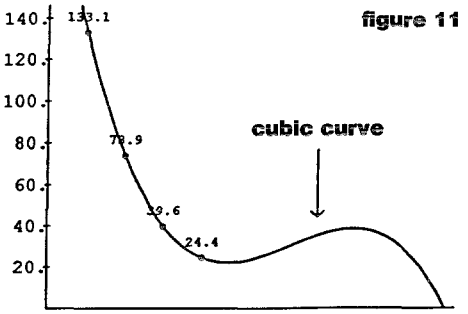
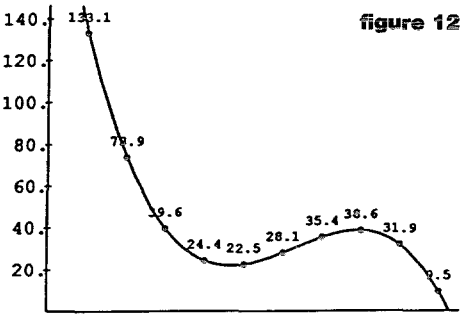
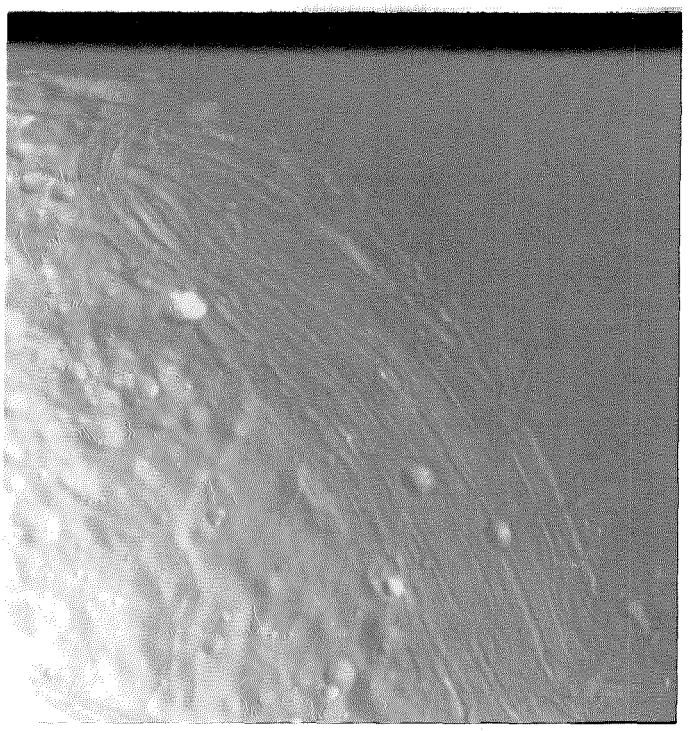


figure 12



With four redundant numbers protecting three pieces of data, we can again correct any pattern of two errors. For example, suppose the parabolic codeword [3.6, 1.2, 2.2, 6.6, 14.4, 25.6, 40.2] were received as [11.6, 1.2, 2.2, 6.6, 3.3, 25.6, 40.2], with errors in the first and fifth positions. To correct the errors, we'd plot the seven received numbers and look for a parabola connecting five of them (figure 9). In this case, it's a little difficult for us humans to see the pattern, but the decoder (computer) doesn't have any trouble, and finds the "majority parabola" immediately (figure 10). Two of the points don't lie on the parabola, so the decoder moves them back on, thereby correcting the errors. Again, with only four redundant numbers, this particular scheme can correct only two errors; to correct more errors, more redundancy is needed, the general rule being that two redundant numbers are needed for each error to be corrected.

What if we wanted to send four numbers at once? Just as two points determine a straight line, and three points determine a parabola, four points determine a *cubic curve* (figure 11). If we want to protect the four numbers from *three* errors, say, then according to Reed and Solomon, we need to choose *six* more points on the curve, thereby producing the 10-number codeword [133.1, 73.9, 39.6, 24.4, 22.5, 28.1, 35.4, 38.6, 31.9, 9.5] (figure 12). When this codeword is received, the decoder plots the points, looks for a cubic curve that goes through at least seven of them, and moves the errant points back onto the cubic, thereby correcting the errors.



Reed-Solomon codes made it possible for Voyager 2 to send back pictures such as this one of geologic details on Miranda, one of the Uranian moons.

On opposite page: Irv Reed (right) and Gus Solomon, who invented their error-correction system in 1960.

Uranus is 2 billion miles away, and Voyager's transmitting power is only 20 watts. That's weaker than the lightbulb in your refrigerator.

This "theoretical" discussion of Reed-Solomon codes gives a pretty accurate idea of how they work, but it's unrealistic in a number of important ways. As I mentioned earlier, Reed-Solomon codes deal with bytes, which are not quite the same as ordinary numbers; and in real applications, there are many more than two, three, or four pieces of data in each code word.

When I teach my students about Reed-Solomon codes, I try to make the subject more practical by dividing the class into teams and having each team write a computer program capable of implementing a fairly powerful RS code. In the particular code I give them, each codeword consists of 15 data characters protected by 16 redundant characters, where a "character" is one of the 26 letters, A, B. . . . Z, or one of the six additional symbols (space), ", #, $, %, and &, so the students work, in effect, with a 32-letter alphabet. If they want to transmit the 15-letter word RUMPLESTILTSKIN, for example, their program must first compute the 16 characters of redundancy, which in this case turn out to be RASZUOBUOS"&YTJS, so the codeword is

RUMPLESTILTSKINRASZUOBUOS"&YTJS

You may find this codeword unattractive; but to the Reed-Solomon decoder, it's beautiful: the 31 characters (when interpreted as special kinds of numbers) all lie on a 14th-degree polynomial curve. With 16 redundant letters, the codeword can resist any combination of eight or fewer errors. For example, if we change the word to

R MCELIECETSKINRASZUOBUOS"&YTJS

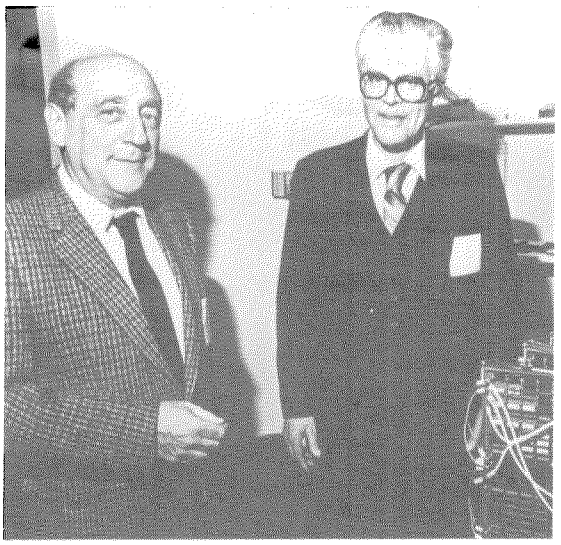
(which I like better), and give it to the decoder, it will see that the pretty pattern has been ruined but will find that 23 of the 31 characters still lie on a 14th-degree curve. (To do this, it uses an algorithm invented by Elwyn Berlekamp, the 20-questions-with-lies guy, who's now a professor at UC Berkeley and president of Cyclotronics, Inc. a company that builds Reed-Solomon hardware.) It will then force the eight offending characters back onto the curve and give us

RUMPLESTILTSKINRASZUOBUOS"&YTJS

again.

There are many applications of Shannon's theorems in general, and Reed-Solomon codes in particular, in today's technology. Error-correcting codes are used in many high-performance military and civilian communication systems. For example, the high-speed modems that today's computers use to talk to each other are made possible by fancy error correction, among other things. One of the most spectacular applications is in the exploration of the solar system. The Voyager 2 spacecraft sent pictures of the planet Uranus back to Earth in January 1986, using what must be the most sophisticated and powerful communication system ever built, on this planet at least. (Uranus is 2 billion miles away, and Voyager's transmitting power is only 20 watts. That's weaker than the lightbulb in your refrigerator.) That communication system has some pretty fancy error correction, which includes a Reed-Solomon code with 223 bytes of data and 32 bytes of redundancy in every codeword. Of course, Voyager's communication system

Reed-Solomon codes, as they're now called, began as only a theoretical curiosity, but today they're probably the most widespread and generally useful error-correcting code system.



depends on a lot of other things too: big and accurate antennas, low-noise receivers, sophisticated transmitters, and much more. Still, without the error-correcting codes, Voyager would have been able to send only about 20 percent of the data that it actually did send.

But applications to communications systems are just half the story. Shannon's theorems have also been applied to the *storage* of information, not just transmission. If you think about it, storage is another form of communication—communication in time rather than space; from *now* to *then* rather than from *here* to *there*. Anyway, there are many storage systems that use Shannon's prescription—computer tapes, disks, and so on. In terms of dollars invested, one of the most widespread applications of Shannon's theorems (via Reed-Solomon coding, in fact) is in data storage. If you own a CD player, then you own a data-storage system that uses Reed-Solomon codes. It's all done using Shannon's prescription, and zeros and ones.

A CD holds up to 74 minutes of music. The music is represented digitally, using lots of zeros and ones. In fact, it takes about 1.5 million bits to represent just one second of music, and more than 6 billion bits are needed for the entire 74 minutes. The bits are stored optically, with tiny "pits" on the mirrorlike surface of one side of the CD. These pits are recorded along a spiral track on the disk, a track that is more than three and a half miles long but only .5 microns wide (.5 microns is approximately the wavelength of green light; that's why light is scattered into a rainbow by the surface of the CD). The pits range from 0.9 to 3.3 microns in length. Such tiny features are quite susceptible to errors; fingerprints, dust, dirt, abrasions, and manufacturing irregularities can all cause problems. So an error-correcting code is used on these disks, and it turns out to be a Reed-Solomon code, in which the bits of information were first blocked into eight-bit bytes. The details are a little complicated (the industry's acronym is "CIRC," for "Cross-Interleaved Reed-Solomon Code") but to protect the 6 billion bits on the disk, another 2 billion error-correcting bits are added, so that fully 25 percent of the bits on the disk are for error correction. Your CD player at home contains a very sophisticated Reed-Solomon decoder, which processes about 2 million coded bits per second.

The result of all this is that a CD is remarkably resistant to errors. I have heard that you can actually drill holes in a CD and it will still play, and I know (since I've done it myself) that you can deliberately scratch one with a paper clip without losing any music. And because of the error correction the music you hear from a scratched disk isn't merely *almost* as good as the original; it's *exactly* as good as the original. Of course, if you get carried away and overdo it, you might really wreck your favorite CD (which is unfortunately what happened to my poor Buddy Holly CD when we shot the photo on page 26). Coding to combat malicious mischief is beyond the scope of this article. □

Bob McEliece has been professor of electrical engineering at Caltech since 1982. He's also an alumnus (BS 1964, PhD 1967), as is Irv Reed (BS 1944, PhD 1949). From 1967 to 1978 McEliece worked at JPL as supervisor of the information processing group in the communications research section, and then became professor of mathematics at the University of Illinois at Urbana-Champaign before returning to his alma mater. McEliece is especially well known for applications of discrete mathematics to various problems in communication theory.

This article was adapted from an April Watson Lecture featuring considerable audience participation. A good time was had by all, thanks to McEliece's talent for making error-correcting codes a lot of fun. Both Reed and Solomon were in the audience at Beckman Auditorium.

If you haven't already peeked, turn the page for the answers to the problems posed in the article.

Answers: Safety in Numbers

page 27

TIARA
ADAPT
TREND
NERVE
SPLIT

page 28

One possibility for six variations on a five-letter word was submitted by Paul Carpenter of Burbank after the Watson Lecture:
STA?E

STAGE
STAKE
STALE
STARE
STATE
STAVE

Doublets:

LEAD
LOAD
GOAD
GOLD

page 29

The correspondence between the eight patterns of YES–NO answers to the three questions, "Is it A, B, C, or D?" "Is it A, B, E, or F?" and "Is it A, C, E, or G?" is as follows:

A:	YES	YES	YES
B:	YES	YES	NO
C:	YES	NO	YES
D:	YES	NO	NO
E:	NO	YES	YES
F:	NO	YES	NO
G:	NO	NO	YES
H:	NO	NO	NO

To solve the 20-questions-with-lies problem, it's best to think of each of the nine answers as a vote against some of the letters. For example, the first answer counts as one vote against each of the letters A, B, C, and D; the second answer is a vote against the letters C, D, G, and H; and so on. In this way, we can calculate a little table, giving the votes against each of the eight letters:

A: 1, 3, 5, 6, 7
B: 1, 3, 4, 6, 7
C: 1, 2, 3, 4, 6, 7
D: 1, 2, 4, 6, 7
E: 3, 6, 7
F: 5, 8, 9
G: 2, 4
H: 2, 4, 6, 7

This table shows that, for example, A has 5 negative votes; in other words, if the letter really were A, then I lied 5 times. Similarly, if it were B, I lied 5 times; if it were C, I lied 6 times, etc. But since I agreed to lie at most twice, and all letters but G have 3 or more negative votes, the letter must have been G, and I must have lied on answers 2 and 4. (Notice also that all 9 questions were needed, since after 8 questions, both F and G were still in the running.)

page 33

If [3.6, 5.9, 8.2, 10.5, 12.8, 15.1] is sent, but received as, say [3.6, 9.1, 8.2, 10.5, 6.4, 5.5] (with errors in positions 2, 5, and 6), then four of the points (9.1, 8.2, 6.4, and 5.5) lie on a straight line, but not the original straight line! (figure A1). But the decoder will have no way of knowing this, and so will move the two points that are off the line back on (figure A2) thereby producing the "codeword" [10.0, 9.1, 8.2, 7.3, 6.4, 5.5] and reporting that the transmitted information was [10.0, 9.1].

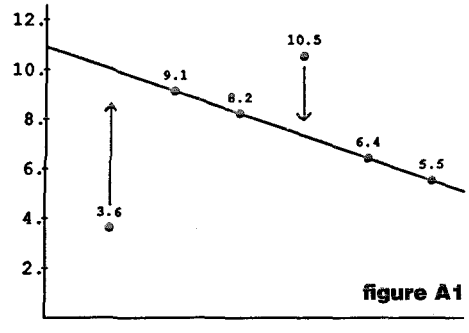


figure A1

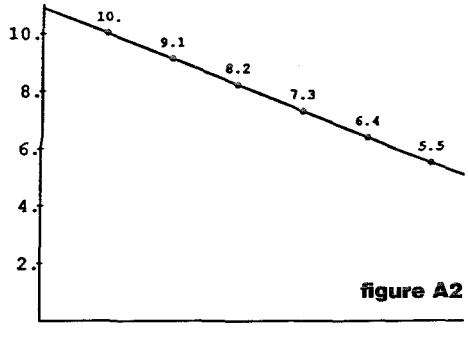


figure A2